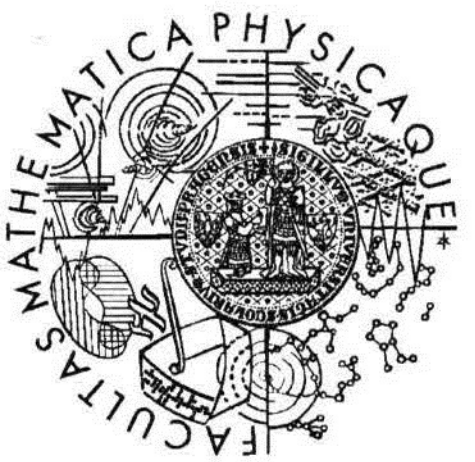
Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

DIPLOMOVÁ PRÁCE



Tomáš Kovařík

Optimalizace exekučního plánu v PostgreSQL

Execution plan optimization in PostgreSQL

Katedra softwarového inženýrství

Vedoucí diplomové práce: Mgr. Július Štroffek
Studijní program: Informatika, obor softwarové systémy

2009

Na tomto mieste by som chcel poďakovať Júliusovi Štroffekovi za vedenie mojej diplomovej práce, cenné rady a predovšetkým za trpezlivosť, ktorú prejavil počas prípravy práce.

Prehlasujem, že som svoju diplomovú prácu napísal samostatne a výhradne s použitím citovaných zdrojov. Súhlasím so zapožičiavaním práce a jej ďalším zverejňovaním.

V Prahe dňa 17.4. 2009 Tomáš Kovařík

Contents

Název práce: Optimalizace exekučního plánu v PostgreSQL
Autor: Tomáš Kovařík
Katedra (ústav): Katedra softwarového inženýrství
Vedoucí diplomové práce: Mgr. Július Štroffek
e-mail vedoucího: julo@stroffek.net

Abstrakt: V predloženej práci študujeme optimalizáciu dotazov v databázových systémoch. Dnešné relačné databázové systémy používajú pre zadávanie dotazov jazyk SQL, ktorý popisuje aké dáta majú byť vybrané z databáze ale nešpecifikuje ako sa má zadaný dotaz vykonať. Optimalizácia databázových dotazov spočíva v nájdení takého sposobu vykonania dotazu, ktorý spotrebováva čo najmenej systémových zdrojov. Pri vykonávaní dotazu je kľúčové poradie, v ktorom sú jednotlivé tabuľky spájané, pretože operácia spojenia tabuliek je typicky veľmi nákladná. Práca sa zaoberá predovšetkým optimalizáciou dotazov s veľkým množstvom tabuliek, pretože u nich nie je možné vyskúšať všetky možné usporiadania tabuliek. V práci prezentujeme niekoľko algoritmov pre hľadanie najlepšieho poradia zjednotenia tabuliek a 6 z týchto algoritmov je implementovaných ako moduly databázového systému PostgreSQL. Práca taktiež prezentuje experimentálne výsledky týchto algoritmov.
Klíčová slova: optimalizácia dotazov, relačné databáze, usporiadanie zjednotení, PostgreSQL

Title: Query execution plan optimization in PostgreSQL
Author: Tomáš Kovařík
Department: Department of Software Engineering
Supervisor: Mgr. Július Štroffek
Supervisor's e-mail address: julo@stroffek.net

Abstract: In the presented work we study optimization of queries in database systems. Current relational databases use SQL language for entering queries. The query in the SQL language describes data which should be fetched from the database, but does not specify how the query should be executed. The goal of the database query optimization is to find a way to execute the query so that it consumes the least of system resources. The key part of query execution is finding optimal ordering for joining relations, because join is one of the most expensive operations. In this work we focus on optimization of queries with large number of tables, since for these queries we cannot search all possible orderings of relations. We present several algorithms for finding optimal join ordering and implement 6 of these algorithms as modules of the PostgreSQL database system. Experimental results and comparison of the algorithms are also a part of this work.
Keywords: query optimization, relational databases, join ordering, PostgreSQL

Chapter 1

Introduction

Storing and accessing data are functions that are performed by almost all computer systems and applications from small websites to large corporate information systems. Relational databases have become a standard for storing large amounts of structured data and although new approaches such as object oriented databases are developed, relational model is still the most popular.

Typical way of accessing data in today's relational databases is using the Structured Query Language, or SQL. Due to the fact that SQL queries are stated in a non-procedural manner, it is the role of the database system to find the best way how to retrieve the requested data from the database. This problem has been the target of research since the first relational database systems were introduced, and is still attracting a lot of attention. With the ever increasing amount of data generated every day, the efficiency of performing queries on the data stored in the database systems becomes critical.

In this work we focus on database queries involving a large number of tables and study different ways how to optimize execution plan for such queries. Special attention is given to determining optimal join ordering, since the join operation has typically a very high evaluation cost and can drammatically influence performance of the whole system. Although it might seem that queries involving large number of tables are not so common, there are many cases where large queries are involved. A good example where complex queries are usually encountered is applications serving as a front-end for the database which automatically generates queries based on user requests.

The goal of this work is to describe, implement and test different algorithms for optimization of query execution plans. The studied algorithms are well known and used in many areas. However, their adoption in database management systems is limited. The implementation and testing is done using an open source database management system called PostgreSQL.

The text is divided into several parts. The first part describes the prob-

lem of query optimization and how it can be addressed. It includes general infomation about database optimizer design and description of algorithms used for execution plan optimization. Second part concentrates on the PostgreSQL database system and describes the query processing in this system in more detail. In the third part we present implementation of different algorithms for query optimization and finally, in the last part, the experimental results of studied algorithms are given.

Chapter 2

Query optimization

Query processing in database systems is usually divided into three main stages: parsing, optimization and execution, see Figure 2.1. The parsing stage, often called *query parser*, is responsible for transforming query from its original textual representation into basic data structures called a *parse tree*. After the query is parsed, it is given to the *query optimizer* for optimization. This stage is sometimes also called *query planning*. The goal of the optimizer is to find a good (or possibly the best) execution plan for the given query. The last part of the system, called *query executor* takes the execution plan and performs all of the specified operations in order to obtain data from the database.

One of the hardest part of query processing is the optimization stage. Relational database query can be represented as an expression in relational algebra using different relational operators. Due to the fact that one relational algebra expression can have multiple equivalent forms, we have multiple ways how to represent a query. Furthermore, there are usually several ways how to perform the operations specified by relational operators. When executed by the database system, each operation in the the query has different cost, which can be expressed for example as a time required to get data from the database. Optimizing the query therefore involves two basic steps:

1. Enumerating different ways how to evaluate the expression in relational algebra

2. Estimating the cost of these evaluations and choosing the one with the lowest cost.

This chapter will first describe how a query can be represented in relational algebra and how different equivalent representations can be used to simplify the query. Then the space of query execution plans will be introduced followed by the ways how to estimate the cost of these plans. Finally, different algorithms for finding the best query execution plan will be described.

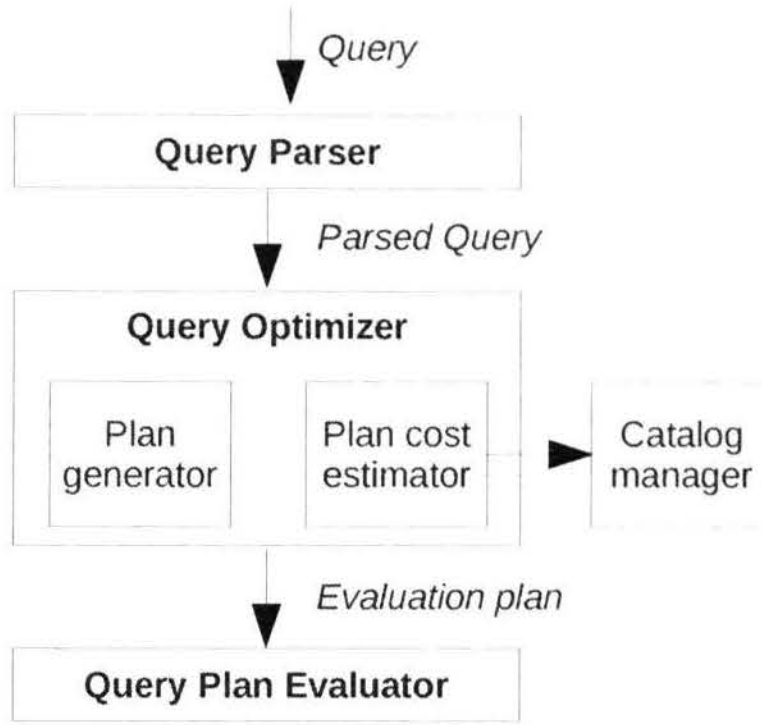


Figure 2.1: Overview of query processing in RDBMS. Source: [12]

2.1 Query representation in relational algebra

Relational algebra is a formal query language, originally proposed by T. Codd as an "algebra on sets of tuples that could be used to express queries about those tuples" [5]. Queries in relational algebra are composed of relations using unary and binary relational operators where both input and output of the operator are relations. To understand how the queries can be represented in relational algebra, several relational operators have to be defined.

2.1.1 Relational algebra operators

The following relational operators are neccessary to describe transformations used by the query optimizer. Full description of relational algebra and it's operators can be found in Molina, Ullman and Widom [5].

Selection (σ) - selection operator specifies the tuples to retain through a selection condition. Applied to a relation R, it produces new relation with a subset of R's tuples. For example, expression $\sigma_{size>5}(R)$ defines a relation containing only those tuples from R with the value of attribute *size* greater than 5.

Projection (π) - projection operator extracts only specified columns from a relation. Applied to a relation R, it produces new relation with only some of R's columns. For example, expression $\pi_{name,age}(R)$ defines a new relation with columns *name* and *age* from relation R.

Cross product ($\times$) - Cross product (often called Cartesian product) of relations R and S ($R \times S$) returns a relation whose schema contains all the fields of R (in the same order as they appear in R) followed by all the fields of S (in the same order as they appear in S). The result contains the concatenation of tuples r and s for every pair of tuples $r \in R, s \in S$

The basic relational operators can be used to define other operations, especially different kinds of joins, which are used to combine information from two or more relations. Two of the most common types of join operations are:

Condition join ($\bowtie_c$) - Condition join (sometimes also called Theta-join) is the most general version of the join operation. The result of the condition join of relations R and S is constructed by taking the product of these relations and selecting only the tuples which satisfy the join condition. The operation can be defined using cross product operator and selection operator as:

$$R \bowtie_c S = \sigma_c(R \times S)$$

Natural join ($\bowtie$) - Natural join is a special case of join operation, where conditions of equality are specified on all fields having the same name in both relations. In other words, natural join of relations R and S puts together only those tuples from R and S that agree in all attributes common to schema of R and schema of S. The new relation created by natural join has all the attributes in the union of schemas R and S.

In addition to these join operations, there are several other operators in so called *extended relational algebra*, such as duplicate elimination, aggregation operators, grouping, sorting and outerjoins. Out of these operators, the most important for query transformation are *outerjoins*. When using traditional join operator, we often find a situation where tuples of one relation do not match any tuple of the other relation on the columns specified by a condition. In this case, the data of these tuples are not present in the result of the join. To include such tuples in the result the outerjoin has to be used. When joining relations R and S using outerjoin, first a natural (or condition) join is executed, then all tuples of R that did not match any tuple of S are added to the result and then all tuples of S that did not match any tuple of R are added. The tuples that were added on top of the result of natural (or condition) join have to be padded with a special *null* value, because they have no corresponding values in the other table. There are three types of outerjoins:

Left outer join - When joining relations R and S using left outer join ($R \bowtie_{LOJ} S$), the result contains the set of tuples returned by ordinary join *and* all the tuples of R which do not have any matching tuples in S (for the given join condition) padded with null values.

Right outer join - The result of joining relations using right outer join ($R \bowtie_{ROJ} S$) is similar to the result of left outer join, except that all tuples of S which do not have any matching tuples in R are added.

Full outer join - Full outer join ($R \bowtie_{OJ} S$) combines left and right outer joins returning all tuples of R without matching tuples in S plus all tuples of S without matching tuples in R, in addition to the result of an ordinary join.

2.1.2 Relational algebra equivalences

As it was pointed out at the beginning of this chapter, relational algebra expressions can have several equivalent forms. To use exact terms, "two relational algebra expressions over the same set of input relations are said to be equivavlent if they produce the same result on all instances of the input relations" [12]. The reason why relational algebra equivalences are important for query optimization is that they allow transformation of the database queries into the form that will be the cheapest to execute.

Equivalences involving selections

Two important equivalences involve selection operation. The first one is called **cascading of selections** and allows us either to combine several selections into one selection operation or split the selection condition consisting of several clauses to a number of smaller selections. Using the notation of relational algebra, cascading of selections is written as:

$$\sigma_{c_1 \wedge c_2 \wedge \cdots cn}(R) \equiv \sigma_{c_1}(\sigma_{c_2}(\ldots(\sigma_{c_n}(R))\ldots))$$

The second equivalence is the **commutativity of selections** which says that conditions may be applied in any order:

$$\sigma_{c_1}(\sigma_{c_2}(R)) \equiv \sigma_{c_2}(\sigma_{c_1}(R))$$

Equivalences involving projections

As with the selections, we can also cascade projections. **Cascading of projections** means that eliminating columns from a relation one by one is the same as eliminating them all at once[1]:

$$\pi_{a_1}(R) \equiv \pi_{a_1}(\pi_{a_2}(\ldots(\pi_{a_n}(R))\ldots))$$

[1]Each a_i in the expression is a set of attributes of R and $a_i \subseteq a_{i+1}$ for $i = 1 \ldots n-1$

Equivalences involving cross-products and joins

Equivalences involving cross-products and joins are important for query optimization, since joining relations is one of the most expensive operations. Being able to transform the query by reorganizing joins and cross-products has a huge impact on performance of the database system, as was pointed out by Selinger et al. already in 1979 [13]. There are two equivalences that allow us to transform relational algebra expression with joins and cross-products. The equivalences below use natural join but hold for any conditional join, since natural join is just a special case of conditional join.

Commutativity - Thanks to the fact that cross-products and joins are commutative, we can choose which relation in a join will be inner and which will be outer:

$$R \times S \equiv S \times R$$

$$R \bowtie S \equiv S \bowtie R$$

Associativity - Associativity of cross-products and joins means that regardless of the order in which the three relations are considered, the result is still the same:

$$R \times (S \times T) \equiv (R \times S) \times T$$

$$R \bowtie (S \bowtie T) \equiv (R \bowtie S) \bowtie T$$

Due to the fact that cross-products and joins are both associative and commutative, we can join any two of the relations, then join the third relation and the result will be the same:

$$R \times (S \times T) \equiv (T \times R) \times S$$

$$R \bowtie (S \bowtie T) \equiv (T \bowtie R) \bowtie S$$

Equivalences involving selections, projections and joins

Certain equivalences involve multiple different operators. When working with selections, projections and joins, we can do the following:

Commute a selection with a projection. This can be done only in case the selection operation involves only attributes retained by the projection.

$$\pi_a(\sigma_c(R)) \equiv \sigma_c(\pi_a(R))$$

Commute a selection with a cross-product or a join . This can only be done if the selection condition involves only attributes of one of the arguments to the cross-product or join (in the example attributes in c must appear only in R, not in S)

$$\sigma_c(R \times S) \equiv \sigma_c(R) \times S$$

$$\sigma_c(R \bowtie S) \equiv \sigma_c(R) \bowtie S$$

Commute a projection with a cross product or join

$$\pi_a(R \times S) \equiv \pi_{a_1}(R) \times \pi_{a_2}(S)$$

$$\pi_a(R \bowtie_c S) \equiv \pi_{a_1}(R) \bowtie_c \pi_{a_2}(S)$$

where a_1 is the subset of attributes in a that appear in R, and a_2 is the subset of attributes in a that appear in S. For the second equivalence to be valid, the join condition c must involve only attributes retained by the projection and every attribute mentioned in the join condition must appear in a.

Equivalences involving outer joins

Outer joins handling is more complicated, since outer joins are not commutative and associative as regular joins and cross-products. However, there are some cases when queries using outer joins can be transformed. Examples of these transformations will be given only for left outer joins because right outer joins can be easily transformed into left outer joins by swapping the input relations. Full outer joins do not allow any transformation at all.

$$(R \bowtie_{LOJ_{RS}} S) \bowtie_{RT} T \equiv (R \bowtie_{RT} T) \bowtie_{LOJ_{RS}} S$$

$$(R \bowtie_{LOJ_{RS}} S) \bowtie_{LOJ_{RT}} T \equiv (R \bowtie_{LOJ_{RT}} T) \bowtie_{LOJ_{RS}} S$$

$$(R \bowtie_{LOJ_{RS}} S) \bowtie_{LOJ_{ST}} T \equiv R \bowtie_{LOJRS} (S \bowtie_{LOJ_{ST}} T$$

Expression $R \bowtie_{LOJ_{rs}} S$ in the above formulas means a Left Outer Join of relations R and S with a join condition RS referencing relations R and S. The third identity only holds if condition ST is *strict* for at least one column of relation S, that is it fails for rows of S which have all values null.

2.2 Query Execution Plans

Job of the query optimizer is to prepare the best execution plan for the query executor. The execution plan can be viewed as a sequence of operations that have to be executed in a given order with given inputs. For every query

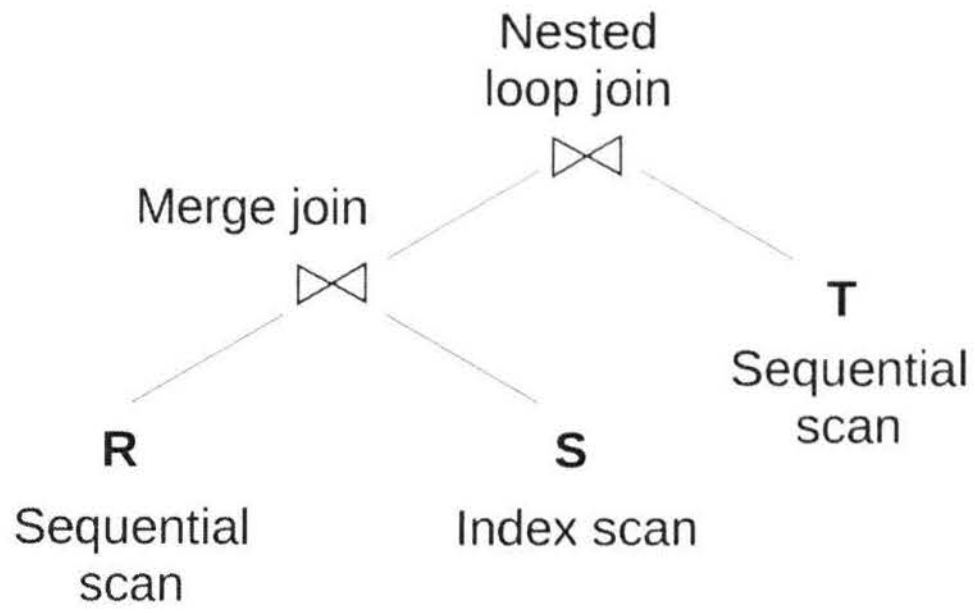


Figure 2.2: Query execution tree

there is typically a huge number of alternative execution plans. One source of alternative plans was described in the previous section which shown that the order of certain operations can be changed thanks to relational algebra equivalences. Another source of alternative plans are methods for implementing relational algebra operators.

Typically there are several algorithms that can be used to implement a relational operator and their use depends on circumstances such as cardinality of the input relations, presence of an index on a given column, available memory and others. For example, the selection operator can retrieve tuples either using a sequential scan or an index scan. A sequential scan of the whole table has to be performed in cases when the input relation is not sorted and there is no index. Index scan can utilize an index to access only those parts of the relation containing tuples that should be selected. Join operation can be implemented either by nested-loop join, merge join or a hash join. Further description of these algorithms can be found in Ramakrishnan, Gehrke [12].

All of these methods of implementation are called *physical operators*. Based on this, a **query execution plan** is defined as an extended relational algebra tree where each node is assigned a physical operator. Query execution plan is sometimes referred to as *physical operator plan* or *query evaluation plan*. In the rest of the text we will use term query execution plan or simply plan. An example of a query execution plan is shown in Figure 2.2. The edges of the tree describe the flow of data between the physical operators in the nodes. The leaves of the tree represent base relations (database tables) with a given physical operator specifying how data in these relations will be accessed. Inner nodes represent the joins of base relations and intermediate relations produced by a join at lower level. In order for the query optimizer to find the cheapest plan, it has to consider all plans that produce the same output.

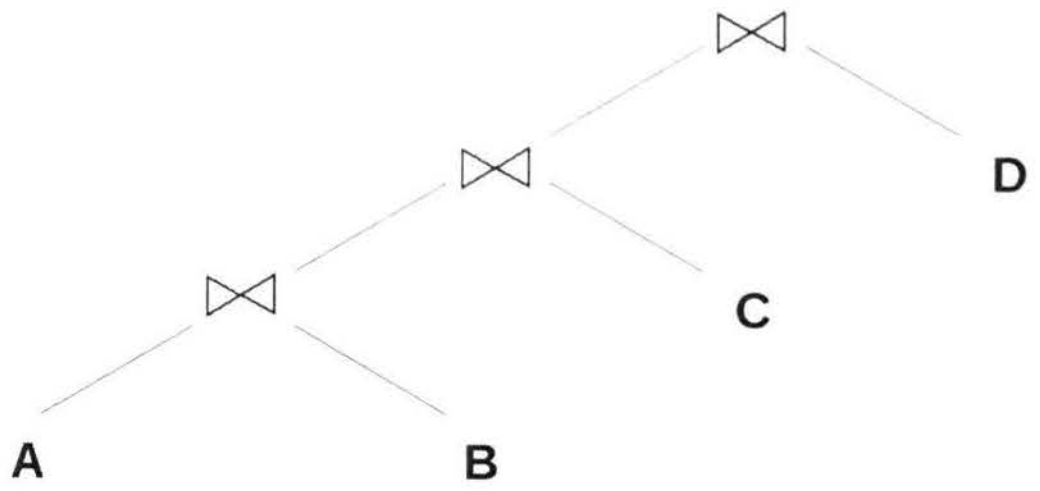


Figure 2.3: Left-deep tree

There are three choices to be made in this process:

1. Choice of the shape of the query processing tree. The shape of the tree defines how the intermediate relations will be produced.

2. Choice of the join ordering. Given a query processing tree of a certain shape, different relations can be assigned to the leaves of the tree causing the relations to be joined in different order.

3. Choice of physical operators to implement operations on relations.

Each of these choices influences the quality of the resulting query execution plan and increases number of possible plans the optimizer has to consider. In the rest of this chapter only the first two choices will be analyzed, because they are the most important for the quality of the query optimizer. The choice of physical operators can be done for a selected query processing tree based on current circumstances, such as relation cardinality and index availability.

Query execution plan is a solution of query optimization problem. All the solutions producing the same result for a given query form a space of valid solutions, sometimes also called a search space, since the optimizer searches this space for the best solution. Certain parts of the search space based on the properties of query execution plans are special. The most common division of query execution plans is based on the shape of the processing tree to left-deep trees and bushy trees.

2.2.1 Left-deep trees

Consider a query joining four relations: $A \bowtie B \bowtie C \bowtie D$. One of the join trees for such a query is in Figure 2.3. The left child of each join node in this tree is called *outer relation* and the right child of the join node is called *inner relation*. The tree in Figure 2.3 is called a **left-deep** tree, since the right child

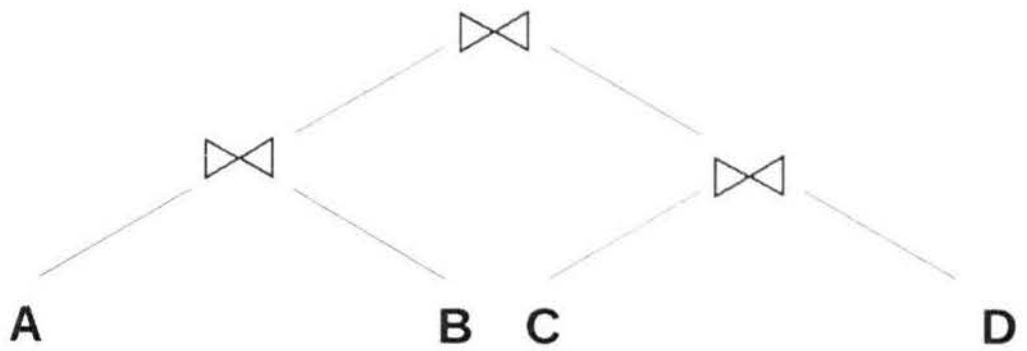


Figure 2.4: Bushy tree

of each node is a base relation. Left-deep trees are an important subset of the optimizer search space and it is not uncommon that database systems search for query execution plans only among left-deep tress (or at least search this subset first). The reason is that good and often the best solutions are believed to exist among left-deep trees and optimizer can utilize pipelining technique[2] on each of the join nodes. Left deep trees substantially limit the size of the search space, since for a fixed number of relations there is only one shape the join tree can have. When considering a left-deep tree for joinig n base relations, there is $n!$ ways to assign relations to leaf nodes of the tree.

2.2.2 Bushy trees

Following the example from previous section, another possible join tree of four relations is shown in Figure 2.4. This join tree is an example of a **bushy tree**. In bushy trees both child nodes of a join node can be a join node (intermediate relation), thus bushy trees include also left-deep trees with all other tree types and do not limit the search space in any way. However, the cardinality of search space with bushy trees is a lot larger than in case of left-deep trees. For n base relations there are $\binom{2(n-1)}{n-1}(n-1)!$ different solutions.

2.3 Statistics and cost estimation

Apart from computing potential solutions, optimizer has to estimate the cost of each solution so that a solution consuming the least system resources (be it CPU time, number of I/O operations or memory used) is executed. This has to be done *accurately* and *efficiently*. The cost estimation must be accurate, because optimization is only as good as its cost estimates [2]. Cost

[2]Pipelining is a technique for passing results from one operator in the query execution tree to another. The main benefit of pipelining is that the intermediate results of an operator do not have to be *materialized* in the main memory or on the disk, thus saving system resources. Further information about pipelining can be found in Ramakrishnan, Gehrke [12]

estimation must also be efficient, since it is repeatedly invoked during query optimization. The basic estimation framework can be done in two steps [2]:

1. Collect statistical summaries of data that has been stored;
2. Given an operator and the statistical summary for each of its input data streams, determine:
   (a) The statistical summary of the output data stream
   (b) The estimated cost of executing the operation

2.3.1 System catalogs and statistical summaries

Every relational database system stores not only data in the tables, but also information *about* the data it contains often called metadata. The metadata are typically stored in a collection of relations called **system catalog**[3]. Information stored in the system catalog is extensively used during query optimization and therefore it is important to know what kind of metadata are collected by the system.

First and foremost, system catalog contains systemwide information, such as size of the buffer pool and page size. There is also information about relations, indexes, views, users and their rights and many more. For instance, a catalog with information about relations might contain items like the relation name, file name where the relation data is stored, file structure, attribute names and their types, names of the indexes and integrity constraints on the relation and others. However the most important metadata for the purpose of query optimization are statistics about relations and indexes. Usually the following statistical information is stored in system catalog:

Cardinality - The number of tuples in each relation, which is used for example to determine the cost of the data scans and joins.

Size - The number of pages used by each relation, which is important for estimation of memory requirements and also for determining I/O cost of accessing the data.

Data distribution - Information about most common values in the column and their frequencies, fraction of missing (null) values and histograms.

Index information - Values such as index cardinality (number of distinct key values), index size (number of pages used), index height (number of nonleaf levels) and index range (minimum and maximum key value)

[3]Some database systems use different name for matadata, such as *catalog relations* or *data dictionary*

Many systems use histograms for storing infomation about data distribution on a column. Although histograms have been used as a visual aid to statistical approximations for quite some time, they first appeared in database systems in 1980.[4] In terms of database systems, histogram divides values in a column into k mutually disjoint subsets called *buckets* and approximates the frequencies and values in each bucket in a common fashion [7]. Number of buckets is usually a constant determining the degree of accurracy of the histogram. The definition of the histogram does not specify how the frequencies and values should be approximated and there are several approaches that can be used:

Value approximation - This says how the data are approximated within a bucket. There are two common approaches, *continuous value assumption* and *uniform spread assumption*. Both of them assume that values are uniformly placed in the range covered by the bucket and uniform spread assumption records also the number of these values in the bucket.

Frequency approximation - This says how frequencies of values are approximated inside a bucket. The dominant approach here is a *uniform distribution assumption* where the frequencies are assumed to be the same and equal to the average of the actual frequencies.

Another important characteristric of a histogram is the way how the data is partitioned and how the buckets are defined. The first histograms used in database systems were called *equi-width* histograms and they divided the data into ranges of equal length (i.e. the spreads of values in each bucket were equal). The next generation of histograms used a different approach and defined buckets so that there was roughly the same number of tuples in each bucket. These are called *equi-depth* histograms and although other types of histograms have been propopsed, many systems use equi-depth histograms to this day. Detailed classification of histograms and their history can be found in [7]

Data stored in histograms is used during query optimization for selectivity estimation. Based on the selectivity of the operations the query is composed of, optimizer decides how to implement those operations, whether the data can be stored in the main memory and what amount of data will flow from one operation to another. Although basic histograms provide good information about data distribution in one column, they fail to provide information about correlation among multiple columns. To capture correlation between columns, a joint distribution of values is necessary. This can be achieved, for example, by using multi-dimensional histograms, which are, however, more complicated.

[4]The first proposal to use histograms can be found in the PhD thesis of R. Kooi [8]

2.3.2 Cost computation

The goal of cost computation step during query optimization is to determine the cost of the operation in terms of CPU time, the number of I/O operations or, in case of distributed systems also the communication cost. Apart from physical and statistical properties of the data, buffer utilization is also important, because the availability of data buffers influences the decision on how the operation will be executed. Accurate cost estimation is one of difficult open problems in query optimization.

2.4 Join enumeration algorithms

In section 2.2 we have described the search space for query optimization characterized by the shape of the query tree, join ordering and available physical operators. Section 2.3 discussed options for estimating the cost of query execution plans (i.e. solutions in the search space). This section brings together search space definition and cost estimation by looking at algorithms used to search for the cheapest solution. Join enumeration algorithms have two basic properties influencing their quality:

- Cost of the found solution
- Amount of time required to find the solution

These properties have opposite effects on the overall quality of the algorithm. The lower the cost of the solution, the longer the time required to find it. The following section will look at several groups of enumeration algorithms and give a description of representatives of each group.

2.4.1 Deterministic algorithms

Given a query, algorithms in this group always construct the same solution. Two examples of deterministic algorithms will be presented. The first is a dynamic programming algorithm for finding optimal solutions by exhaustive search. Second is a greedy algorithm, which, however, does not guarantee the optimal result.

Dynamic programming algorithm

This is a classical algorithm for join order optimization originaly developed for IBM System-R in 1979[5] and is a standard for query optimizers in relational database systems even today. Due to it's first use in System-R, dynamic programming algorithm for optimizing join order is sometimes also

[5]The original description can be found in Selinger et. al. [13]

called *System-R* algorithm. Originaly, the algorithm was developed to search for solutions only among left-deep trees; however it has been extended to search the whole search space. Algorithm constructs the optimal solution by building partial solutions of increasing size, starting with all possible scan nodes for all relations in a query[6]. Next, only the cheapest partial solution is retained for all groups of equivalent solutions. Two partial solutions are considered equivalent if and only if they join the same set of relations and the sort order of the partial result is the same. Those partial solutions that were retained are an input for the next iteration. In the *k-th* iteration we have a set of *k*-relation partial solutions and we construct *k+1*-relation solutions by joining all base relations to all *k*-relation solutions and retaining only the cheapest ones for each equivalent group. In the last iteration, when we construct *n*-relation solutions from *n-1*-relation solutions we have at least one optimal solution for joining all the relations in the query.

The original algorithm included an extra heuristic which defers including cross products of tables as long as possible. The reasoning behind defering cross products is, that an optimal solution is unlikely to put cross products early in the execution since they significantly increase the size of partial results. Although with this prunning the algorithm is not guaranteed to return optimal solution, it is a standard to include it in the implementation of the optimizer.

The major disadvantages of the dynamic programming algorithm are it's exponential running time, depending on number of relations in the query and high memory consumption for storing partial solutions. This makes it generally unusable for queries with more than 15 relations[7].

Since it's inception, multiple enhancements of dynamic programming algorithm were proposed. Examples of the recent ones can be found in Neumann, Moerkotte (2006) [10] or Neumann, Moerkotte (2008) [11]. These enhancements typically allow for larger queries to be optimized by dynamic programming algorithm, but often place restrictions on query structure and composition (such as queries without cross products). Interesting results of dynamic programming were presented in Lohman et. al [6] where the authors took advantage of increasing number of CPU cores used in contemporary systems and parallelized the execution of dynamic programming algorithm. Using a system with 8 CPU cores they were able to optimize queries of up to 20 relations.

---

[6]Given that there is an index on attribute A of relation R, two items are added to the initial list of solutions: index scan of relation R based on index on R.A; and a sequential scan of the whole R

[7]This number of course depends on type of the query and hardware used, but even increasing hardware performance cannot beat exponential running time

Greedy algorithm

Greedy algorithm is any algorithm which makes locally optimal decision in each of its steps in the hope that it will lead to a global optimal solution [14]. In other words, greedy algorithm picks the solution which looks the best at the moment and never reconsiders any previous choices. Examples of greedy algorithms are Kruskal's algorithm for finding minimum spanning trees or Dijkstra's algorithm for finding shortest paths from a single source. Although in these cases the algorithm finds the best solution, it is not so for other problems. In order for a greedy algorithm to find globally optimal solution, the underlying problem has to have *optimal substructure*, i.e. an optimal solution to the whole problem must contain optimal solutions to the sub-problems [14]. Applied to the join ordering problem in query optimization, greedy algorithm constructs the solution incrementally, adding one relation in each step. The relation to be added is chosen so that the new solution has the lowest cost. Due to the nature of the join ordering problem, greedy algorithm does not always achieve optimal solution. Despite this deficiency, it is used because of the low execution time.

2.4.2 Randomized algorithms

Algorithms in this class do not always return the same result because they try to find the solution by randomly moving throught the search space. Key characteristic of randomized algorithms is that they define a set of *moves* in the search space which represent edges connecting individual solutions[8]. Randomized algorithms typically perform a walk along these edges visiting certain number of solutions and returning the best solution of those visited. An apparent drawback of randomized algorithms is that the best solution might not be among those visited. On the other hand, the time spent on searching for the best solution is dramatically lower than for exhaustive search and therefore these algorithms can be used even for queries involving large number of relations.

Enumeration algorithms often use the terms *neighbour* and *neighbourhood* of a solution. In order to clearly understand the algorithms, we need to define these terms as well as the *move* (also called a *transformation*) from one solution to another. There are two commonly used ways how to define a move:

Swap - the target solution is generated by simply swapping 2 relations in the join order of an existing solution.

[8]The search space can be viewed as a graph where every solution (i.e. query execution plan) represents a vertex and move from one solution to another represents an edge connecting the two vertices

3Cycle - the target solution is generated by cycling 3 relations in the join order of an existing solution (first relation takes the place of the second relation, second relation takes the place of third relation and third relation takes the place of the first relation)

Using these definitions, a *neighbour* of a solution is a solution which can be reached by performing exactly one move and *neighbourhood* of a solution is a set of all neighbours (all solutions reachable by exactly one move).

Random sampling algorithm

Random samling algorithm is based on an idea that if the search space contains certain number of good[9] solutions, we should be able to find one of such solutions by repeatedly picking random solutions. The algorithm and its implementation are therefore pretty simple - generate n random solutions and choose one with the lowest cost. However, there are several theoretical problems. First of all, the assumption that the search space contains enough good solutions might not be valid. It has been shown that the proportion of good solutions in the search space decreases quickly with the number of relations in the query [4]. This could be dealt with by increasing the number of random solutions tried, although a reasonable compromise has to be found so that the algorithm doesn't run too long. Another problem of random sampling is, that generating random solutions which are also valid for the given query is not an easy task[10]. The quality of the results of this algorithm therefore depends on the cost distribution in the search space and the ability to generate truly random solutions.

Hill-Climbing algorithm

Hill-climbing algorithm is one of the simpliest randomized enumeration algorithms. Since the algorithm considers only a small subset of the search space, the quality of the results is limited but it is one of the fastest ways how to find a join ordering. The algorithm chooses a random starting point and searches a given number of neighbour solutions. If there is a better solution, it makes the move and starts searching the neighbours again. Due to the fact that there is only one starting point, algorithm can reach a local minimum from which it cannot escape. Since there is a large number of neighbours for any single solution, it doesn't search the whole neighbourhood of

[9]There is no widely used definition of a good solution. Good solution can be one with cost lower than two times the cost of optimal solution[4].

[10]The problem lies in one-to-one mapping of the query trees to combinatorial structures used to represent them [4]. Because of this, the algorithm is also called *quasi-random sampling*.

the solution but considers only predefined number of random neighbours for each solution.

Iterative improvement

Iterative improvement is an enhanced version of the hill-climbing algorithm. It overcomes the main deficiency of hill-climbing since it doesn't stop searching for the best solution in case a local minimum is reached. Iterative improvement starts with a random solution and then moves to cheaper neighbours the same way as hill-climbing. However, once no cheaper neighbour has been found (local minimum has been reached), it restarts the search from a new random solution. This is either repeated for a given number of starting points or until the time limit set for algorithm is exceeded.

Simulated annealing

When iterative improvement algorithm reaches local minumum, the only way to escape it is to start a new search. In some cases even large number of restarts might not help, especially if the search space contains significant number of high-cost local minima. Simulated annealing algorithm tries to overcome this problem by allowing a move to neighbouring solution with a higher cost. As the name of the algorithm suggests, the inspiration comes from annealing process in metallurgy, where material is first heated and then cooled in order to increase the size of the crystals and reduce number of defects. Algorithm uses also terminology of the annealing process and uses terms like temperature, equilibrium and freezing condition. The algorithm starts from a random solution and operates in stages where each stage is executed with a fixed value of a parameter T called *temperature*. Temperature controls the probability of accepting uphill moves (moving to a solution with a higher cost). This probability is set as $e^{\Delta C/T}$ where ΔC is the difference between cost of the current and the new solution. During one stage, the algorithms moves to the neighbour solutions until it reaches *equilibrium*. Equilibrium is usually defined as executing certain number of stages, such as number of relations in the query or its multiple. Once equilibrium is reached, the temperature is decreased by a certain amount and searching for solution continues. The algorithm finishes once a *freezing condition* is met. For example, it may be frozen if no better solution was found in certain number of temperature reduction cycles.

Two phase optimization

The advantage of simulated annealing is that it thouroughly searches certain part of the search space. However, since it only uses one random start-

ing point, it usually doesn't cover a large enough portion of the search space (compared for example with iterative improvement algorithm). These observations about the nature of iterative improvement and simulated annealing lead to another algorithm for finding the join ordering called Two phase optimization (2PO). This algorithm combines simulated annealing and iterative improvement trying to achieve the advantages of both. The algorithm has two steps:

1. Iterative improvement algorithm is used for a given number of random starting points returning a set of local minuma.

2. The lowest of the local minima from step 1 is used as a starting point for simulated annealing, which is executed with lower initial temperature because only a small neighbourhood of a given local minimum has to be searched[11]

By combining iterative improvement and simulated annealing, larger portion of the search space is covered, the algorithm doesn't get trapped in a local minimum and can thoroughly search part of the space close to the best local minimum.

2.4.3 Genetic algorithms

Genetic algorithms performs a randomized search of the space in principle similar to biological evolution. Although it resembles randomized algorithms by considering only a subset of solutions, the process is different enough to be described separately. This section will describe motivation for genetic algorithms and basic working principles. Detailed discussion of genetic algorithm for database query optimization can be found in Bennet et. al [1].

Natural evolution process is based on the survival of the fittest members of the population who propagate their genetic information to their offsprings, thus giving birth to a new population. Genetic algorithms mimic this process and use similar terminology. Since a single specimen is not so important for evolutionary process, genetic algorithms do not consider one solution at a time but a set of solutions called *population*. Solutions are represented by *strings*, also called *chromosomes*, composed of *characters* called *genes*. For every problem solved by a genetic algorithm there must exist an *encoding* of solutions to character strings. Additionaly, an objective function measuring the quality of solutions has to be defined. Genetic algorithms use term *fitness* to describe quality of a given solution.

[11]Simulated annealing doesn't have to "go too far" because other parts of the search space should be covered by iterative improvement runs in step 1.

Basic algorithm

Genetic algorithm for query optimization is initialized by a population of strings representing random solutions denoted as *zero generation*. New generations are created from the zero generation using the following operations:

Selection - Certain number of the best (fittest) members of the population survives to the next generation.

Crossover - Certain number of the fittest members of the population is combined into an offspring which survives to the next generation

Mutation - Certain number of members of the population is randomly modified (mutated) and propagated to the next generation

This loop is repeated until a stopping condition is met, e.g. a predefined number of generations has been produced, the fittest member of the population reached a desired quality or there was no significant increase in maximum fitness for several generations. The following paragraphs will describe operations used in the algorithm in more detail.

Solution encoding

Encoding defines how a solution is represented using strings. The operations performed by the genetic algorithm are dependent on the solution encoding. We will give two examples of encoding solutions of join ordering problem[12]:

Ordered list - Each solution is represented as an ordered list of leaves of the query tree starting from the leftmost leaf.

Ordinal numbers - Here, the solutions are encoded by a sequence of ordinal numbers of relations in the ordered list of relations. For example given the following query $(R2 \bowtie R1) \bowtie R3$ and a list of relations $[R_1, R_2, R_3]$, the ordinal number are generated this way:

1. Get the first relation in the query (R_2), find it's index in the list of relations (2) and return it.
2. Remove R_2 from the list of relations
3. Repeat previous steps until list of relations is empty

The ordinal numbers for this query are [2,1,1].

[12]For the sake of simplicity, only encoding of solutions representing left-deep trees is described. For the descripion of bushy trees encoding, see [9]

Selection

Selection is used to remove bad solutions from the population. Solutions to survive to the next population are selected randomly with better solutions having greater chance to survive. The chance to survive is typically inversely proportional to the cost of the solution.

Crossover

Crossover operation is used to combine two good solutions and propagate the offsprings into the next generation. The crossover operation is strongly dependent on solution encoding because no gene can be missing or be duplicated in the offspring. There are several ways to perform crossover for each encoding, for example:

Subsequence exchange - Assuming we have a solution encoded with ordinal numbers, subsequence exchange selects a sequence of genes of equal length from each parent and produces offspring by swapping these sequences. If we have parents encoded by strings 2311 and 1211 and select sequences of length 2 starting from the second gene (31 and 21), we get offsprings with strings 2211 and 1311.

Subset exchange - Assuming we have ordered list encoded solution, we have to ensure there are no duplicated or missing characters. When performing subset exchange, select two random subsequences of equal length that consist of the same characters from both parents and swap them. This way, for parents encoded by strings 13254 and 52431 we can choose characters 2, 4 and 5 and swap 254 for 524 producing offsprings with strings 13524 and 25431.

Mutation

Mutation operation is used for introducing properties that are not present in any solution in the population. Typically it is done as random change of a gene or several genes in a solution. In case of ordered list encoding this is not possible because we have to preserve all the genes, therefore mutation is implemented by swaping two random genes in a solution. The frequency of mutations cannot be too high because it would disrupt the process of improving populations from one to another.

Chapter 3

Query processing in PostgreSQL

This chapter describes implementation of query processing in PostgreSQL database system. When describing query optimization, it is neccesary to understand what happens to a query from the moment it is entered by the user until the moment results are returned. In most traditional Relational database management systems (RDMS), the basic structure of a query processor is similar. However implementation and capabilities of query processors differ in various database systems. In this chapter we concentrate on query processing in PostgreSQL database system, which was chosen as a platform for implementation and testing of algorithms for execution plan optimization due to its open nature, high quality documentation and good extensibility.

3.1 PostgreSQL introduction

PostgreSQL, also called Postgres, is an object-relational database management system with rich set of features comparable to other open source and commercial database systems. Apart from the feautures, the strengths of Postgres are especially reliability, good performance characteristics, conformance to SQL standards and extensibility through different modules. Postgres also runs on wide variety of platforms including UNIX and UNIX-like operating systems (FreeBSD, Linux, Mac OS X) and Microsoft Windows operating system.

Another huge advantage of Postgres over other database management systems is its open source nature and the fact, that it is distributed under a very liberal BSD licence[1]. Compared to some other open source databases, there is no single company behind the development of PostgreSQL. Decisions about Postgres and its development are made by an international com-

[1]Full text of PostgreSQL licence can be found at `http://www.postgresql.org/about/licence`

munity of developers, contributors and users. Community and its members also provide support and help via public mailing lists and other communication channels. Thanks to the licensing policy, there is also a number of companies offering professional support services and other products based on Postgres.

3.2 History of PostgreSQL

The roots of PostgreSQL go back to 1970s and a system called Ingres, which was developed at the University of California at Berkley between 1977 and 1985 and later succesfully comercionalized. Development at Berkley continued and in 1986, Professor Michael Stonebraker started working on an implementation of POSTGRES (the name comes from a fraze "post-ingres"). In 1994 a company called Illustra, now a part of Informix, which became part of IBM started offering POSTGRES as a commercial product. However, two graduate students at Berkley continued development of POSTGRES, added support for SQL query language and released the result under the name Postgres95. In 1996, the development left University of California at Berkley and was moved to several programmers from around the world. At that time, the name Postgres95 was changed to PostgreSQL to signify the connection to original POSTGRES and the ability to work with queries in SQL query language. Today, both official name PostgreSQL and its shortened version Postgres (not capitalized) are used to describe the system.

Development of Postgres continued since 1996 with several important releases, which were at first aimed primarily at resolving bugs, providing reliability and decent performance. Later a number of features was added, implementing large parts of SQL standards SQL92 and SQL99 which made Postgres suitable for most applications of database management systems. Today, the most recent released version is PostgreSQL 8.3.7 and development of version 8.4 is underway.

3.3 Architecture of PostgreSQL

Architecture of PostgreSQL is based on a client/server approach. Any client that needs to access data in the database cannot do so directly, but has to connect and communicate with the server process. The model can also be called "process per user", since there is always one client process connected to one server process. In order to accommodate unknown number of clients, there is a master server process called *postmaster* that spawns a new server process every time new client is connected. Synchronization of server processes is done using semaphores and shared memory.

The client/server architecture of PostgreSQL has several advantages. First and foremost, separation of client and server processes helps to ensure security and reliability of the system. Client/server approach also works well in networked environment and allows portability to many operating systems. On the other hand, there are also drawbacks. Because it is necessary for server processes to communicate with each other via shared memory, scalability and spreading one server over multiple computers is limited. Also, spawning of server process for every connection requires certain overhead and can be slower when compared to other possible methods such as multiple program threads inside one process. This becomes a significant issue for client task with very small duration but can be resolved by connection pooling.

3.3.1 Backend structure

Backend is the server process created to serve a connected client. After the connection is established, system enters the following query loop: user types a query which is transmitted by the client to the backend, there is no parsing of query at the client, it is transmitted as a plain text; backend processes the query and returns results to the user, who can enter another query and continue the loop.

Query processing in the backend consists of the following stagesi, which will be described later in more detail:

Parser parses the query and performs syntactic and semantic analysis.

Rewriter - modifies query based on available rewriting rules, such as substition of views.

Planner/Optimizer selects optimal query execution plan.

Executor executes selected plan and returns the results (e.g. rows of the table).

During its execution, backend relies on the information stored in the system catalogs. System catalogs are a place where system stores schema metadata, such as information about tables, columns, indexes etc. PostgreSQL is more catalog-driven, than other similar database management systems and stores also information about datatypes, functions, operators, index access methods etc. System catalogs are regular tables which can be modified using SQL statements, though they should not be modified directly but through specific SQL extensions. The advantage of storing information in system catalogs is the system extensibility – the system can be extended by adding new catalog entries and writing functions implementing the operation.

3.3.2 Parser

The main function of parser is to perform syntactic and semantic analysis of the query and transform it from the textual representation to data structures, which can be then used by the rest of the backend. Operation of the parser is divided into two separate stages: *parsing* and *transformation*.

In the parsing stage, SQL query in its original text form is checked for correct SQL syntax and a parse tree is built. No metadata lookups are done at this stage, therefore no access to the data in the database is necessary.

Second, transformation stage modifies and auguments data structures returned by the parsing stage. This stage does not look at the syntax of the query, but tries to understand what tables, functions and operators are referenced by the query. The output of the transformation stage is called a query tree, which is similar to parse tree produced by the first stage, but includes other information such as data types of columns and results of the expressions.

The reason for separating parsing and transformation stage is to postpone metadata lookups, because they have to be enclosed in the transaction and appropriate locks on database objects have to be obtained.

3.3.3 Rewriter

Rewriter is a part of the backend implementing the PostgreSQL rule system. Rule system (or more precisely query rewrite rule system) is responsible for rewriting queries according to certain rules, which are defined and stored in advance. PostgreSQL uses the rule system to implement database views, however the full potential of the rule system is much bigger than just handling views. Full documentation of the rule system can be found at `http://www.postgresql.org/docs/8.3/interactive/rules.html`.

3.3.4 Planner / Optimizer

Query optimizer, also called query planner, is responsible for generating query execution plan which is used to execute the query. A detailed description of query optimization was given in Chapter 2, therefore, we will only focus on PostgreSQL specific issues.

PostgreSQL performs query transformations based on relational algebra equivalences described in the previous chapter. When full outer join is processed, relations on both sides of the join are optimized separately as if they were in a separate query. If the query contains a subquery, optimizer tries to pull relations from the subquery to the main query so that all relations can be processed together. Pulling subqueries up is not possible if they contain

aggregate functions, GROUP BY or DISTINCT clauses. In that case, subquery is planned separately and the main query works only with the results of the subquery.

To determine optimal join ordering, PostgreSQL uses dynamic programming algorithm for smaller queries and genetic algorithm for larger queries. Both algorithms consider space of bushy trees when searching for optimal ordering. The threshold for using genetic algorithm instead of dynamic programming can be configured in the PostgreSQL configuration file. The default setting is to use genetic algorithm for queries with 12 or more tables.

3.3.5 Executor

Job of the Executor is to take the plan from planner, recursively process it and pull the data from the database. This is done by applying demand-pull recursive mechanism. Every time a plan node is called, it is expected to return next row of its input or report that there are no more rows to return. There are two types of plan nodes – bottom level nodes (leaves of the tree) and upper level nodes (inner nodes of the tree). Bottom level nodes are scans of database tables, such as sequential scan or index scan. If a bottom level node is called during executor run, it returns next row of the scaned table. Upper level nodes are usually join nodes, combining two streams of data, which can be either tables or another join nodes. Requesting a row from the topmost node therefore causes recursive call to all the underlying nodes and produces the final result. To implement all the features SQL language offers, there are also special node types for sorting data and aggregation which operate in a similar manner.

3.4 Comparison with other open source database systems

PostgreSQL is not the only open source database management system available today. Although the structure of query processors is similar in most of the available systems, there are some differences worth mentioning. This part of the text will compare the design of query processing in PostgreSQL with two open source relational databases, MySQL and Apache Derby.

3.4.1 MySQL

MySQL is one of the most popular open source relational database management systems. It is widely used especially for web applications in connection with PHP and other technologies. Unlike PostgreSQL, MySQL is

developed and maintained by a single company, Sun Microsystems. The original company which developed and maintained MySQL was bought by Sun Microsystems in 2008.

This section will first briefly describe architecture and features of MySQL and then will concentrate on optimizer design in MySQL. Detailed information about MySQL can be found in the online documentation available at `http://dev.mysql.com/doc/`. Information in this text describes MySQL version 5.1.

Architecture and features of MySQL

MySQL database system is using client/server model. The server side is performing operations on data and serves connected clients using multi-threaded architecture. An interesting part of MySQL architecture is availability of multiple storage engines performing low level operations with data. Due to the differences in storage engine implementations, some of the features are available only is some engines. The most popular storage engines used in MySQL today are InnoDB and MyISAM and there are several new ones under development.

MySQL offers a set of features sufficient for performing basic database operations typical for many web based applications, however, its application in enterprise environment is limited [3]. Several typical database features, required not only in enterprise environment, were added to MySQL only recently, such as support for triggers, procedural languages or views.

MySQL Optimizer design

MySQL query optimizer is in principle similar to that of PostgreSQL. Given a query processed by the query parser, it tries to perform certain simplifications and query transformation based on relational algebra equivalences. Join order optimization is executed using either exhaustive depth first search algorithm or a greedy algorithm. Both algorithms consider only the space of left-deep trees. Greedy algorithm was added in MySQL version 5.0 and should replace the original algorithm completely in the future. This can be done due to the fact that greedy uses exhaustive search to a certain depth when looking for the relation that should extend the partial solution. In case that the query contains fewer relations than the lookahead depth, exahustive search is performed.

3.4.2 Apache Derby

Apache Derby is a small open source relational database written in the Java programming language. The main advantage of Derby is its small footprint,

only about 2 megabytes. Thanks to its size and the fact, that it is written in a platform independent language, it can run on wide range of devices from cell phones and PDAs to traditional servers, provided that the Java Run-time Environment is available on the given platform. Derby can be either embedded right into the java application using it or it can be run in a more traditional client/server setting.

Derby optimizer

Since Derby is intended for use on devices where other database systems cannot run because of their higher system requirements, it is not expected to execute very complex queries. This also influences the design of the query optimizer. Derby optimizer[2] searches only the space of left-deep trees and uses an exhaustive search for finding the best join ordering, implemented as a depth first search with cost-based search space pruning. The optimizer remebers the best complete solution found so far and immediately rejects partial solutions if their cost is higher than the cost of the best solution.

[2]Detailed description of the optimizer can be found in Derby documentation at `http://db.apache.org/derby/papers/optimizer.html`

Chapter 4

Implementation of Join Order Search in PostgreSQL

The algorithms for join order optimization are not new and their theoretical background was studied for more than 20 years. However, their implementation in today's relational database systems is limited. More advanced relational databases implement some of these algorithms, but a presence of several algorithms in one system, which would allow their comparison, is not typical. There is a number of experimental studies providing implementation and benchmarks of these algorithms, but they are usually done using experimental database systems.

One of the goals of this work is to introduce different join optimization algorithms to a real database system. PostgreSQL has been the database system of choice for several reasons:

- Open source nature of the system.
- Source code and documentation quality.
- Active community able to provide neceassary support.

PostgreSQL source code is released under a liberal open source licence, which makes it easier to study, modify and extend it. Furthermore, the source code is clean and well documented, especially when compared to source code of other open source relational databases such as MySQL or Firebird. This makes modifications significantly easier.

This chapter will describe the development and implementation of the Join Order Search module, data structures and routines from the current PostgreSQL optimizer that were used during implementation of the join enumeration algorithms and several design issues common to all implemented algorithms.

4.1 Pluggable optimizers framework

Current implementation of join enumeration algorithms in Postgres, both dynamic programming and genetic optimizer is tightly integrated into the source code and cannot be easily separated. When thinking about modifying the current implementation, we have considered several options how it could be done. The final solution was to put the new code into a separate module, leaving the existing algorithms as they are and introducing a hook for bypassing the original implementation. This way, only a minor change to the core of PostgreSQL backend was neccessary and enabled us to implement new algorithms without compromising the functionality of the system.

New join enumeration algorithms were put into a separate *contrib module* caled *jos*. Contrib modules in Postgres are a way how to extend core functionality of the system. They are typically different plug-in features and tools that either have limited audience or are too experimental to be included in the core distribution. Once the database system is installed, the user can choose which modules to add to the system.

The modification of the PostgreSQL optimizer was limited to adding a `join_search_hook` variable to file `src/backend/optimizer/path/allpaths.c` and modifying function `make_rel_from_joinlist()` in the same file, so that if `join_search_hook` is defined, a function pointed to by the hook is called instead of standard optimization or genetic optimizer. This change has been already released in PostgreSQL 8.3.

The JOS module consists of several libraries written in C language, each containing an implementation of a single algorithm. With a hook in the PostgreSQL source code, libraries containing an implementation of join ordering algorithm can be loaded into running instance of Postgres using a LOAD command[1]. Once loaded into the backend, the algorithm takes care of join ordering search for submitted queries. In order for PostgreSQL to know how to call the given algorithm, all libraries in JOS module must have a common structure shown in Listing 4.1. There are several things about the common structure of a JOS library worth mentioning:

- Each library must include a Postgres header file, therefore the JOS module must be compiled with a link to PostgreSQL header files.

- `PG_MODULE_MAGIC` macro is a way how to ensure that a library is compatible with a version of Postgres it is running on. This macro inserts a special magic block into the library which is checked once the library is loaded into PostgreSQL

[1]LOAD is a special PostgreSQL command which loads a shared library file into the PostgreSQL server's address space

- `_PG_init()` and `_PG_fini()` functions are called when the library is loaded and unloaded from Postgres. `_PG_init()` function sets the value of a `join_search_hook` to a function implementing given enumeration algorithm. `_PG_fini()` unsets the hook which means that standard optimizer is used.
- The function assigned to `join_search_hook` must have a prototype exactly as the one in Listing 4.1.

Listing 4.1: Common structure of a JOS module library

```
#include "postgres.h"

PG_MODULE_MAGIC;

void    _PG_init(void);
void    _PG_fini(void);

static
RelOptInfo *
dummy_join_search(PlannerInfo *root, int levels_needed, List *initial_rels)
{
        // Here comes implementation of the join enumeration algorithm
}

void
_PG_init(void)
{
        join_search_hook = dummy_join_search;
}

void
_PG_fini(void)
{
        join_search_hook = 0;
}
```

4.2 PostgreSQL optimizer routines and data structures

Custom join enumeration algorithms in the JOS module use a number of routines and data structures from the main optimizer source code. To understand the implementation of the join enumeration algorithms, we will first describe the most important routines and data structures.

4.2.1 Data structures

PlannerGlobal holds global information for a single planner invocation

PlannerInfo contains information for planning a particular Query. PlannerInfo instances are usually called *root* in the optimizer routines. It is also

one of the parameters of every JOS library implementing an enumeration algorithm.

RelOptInfo is an important structure which represents a base relation or a join relation during optimization.

RestrictInfo holds information about WHERE clauses and also join conditions.

Path structures represent a way to generate a relation. There is a different path structure representing each of the more complicated operations, such as *IndexPath* representing index scans, *HashPath* representing hash joins etc. Simple operations which do not require special information, such as sequential scan are represented by a generic Path structure.

EquivalenceClass hold information about a set of values which are equal. Equivalence classes are created based on equality conditions in the query and allow the optimizer to treat all members of an equivalence class together.

The most important data structures for query enumeration are *PlannerInfo*, *RelOptInfo* and *RestrictInfo* and will be described in more detail.

PlannerInfo structure

PlannerInfo contains working state of the planner invocation. The fields of interest for join enumeration are:

simple_rel_array holds pointers to RelOptInfo structures for all base relations

join_rel_list and join_rel_hash contain pointers to RelOptInfo structures for *join relations* (i.e. relations made up of at least two base relations joined together). `join_rel_hash` is used only if there is large number of join relations and it speeds up access to these relations by building a hash table.

left_join_clauses, right_join_clauses, full_join_clauses fields contain lists of RestrictInfo clauses for left, right and full outerjoins, respectively. RestrictInfo structure is described below.

join_info_list contains information about restrictions on relation reordering placed by outer joins, IN and EXISTS clauses. `join_info_list` contains a `SpecialJoinInfo` structure for each one-sided outerjoin and IN

and EXISTS clause[2]. `SpecialJoinInfo` contains primarily information about relations on both sides of outerjoin and is used when deciding wheteher a join order is valid or not.

RelOptInfo structure

RelOptInfo is a structure representing tables and intermediate results during the optimization process. There are two different objects that the RelOptInfo might be used to describe. Firstly, it holds information about *base relations*. In Postgres optimizer, a base relation might be either a single table, result of a sub-select or output of a function. All of these are handled in the same manner. Second object a RelOptInfo can describe is a *join relation*. Join relations represent join of two or more base relations. An important property of RelOptInfos for join relations is that **there is only one RelOptInfo for any set of base relations joined together**[3].

ReloptInfo has the following fields (again, only selected fields important for join enumeration algorithms are described):

relids - Set of base relation identifiers; it is a base relation if there is just one, a join relation if there are more than one

rows - estimated number of tuples in the relation after restriction clauses have been applied

width - average. number of bytes per tuple in the relation after the appropriate projections have been done

pathlist - list of Path nodes, one for each potentially useful method of generating the relation

cheapest_startup_path - the pathlist member with the lowest startup cost, regardless of its ordering

cheapest_total_path - the pathlist member with the lowest total cost, regardless of its ordering

cheapest_unique_path - the pathlist member with the lowest cost that produces output with no duplicates

The following fields are set only if RelOptInfo holds a base relation:

relid - identifier of the base relation

---

[2]In fact, it also contains SpecialJoinInfo structures for full outer joins but since full outer joins cannot be reordered, they are there only to simplify other parts of the code

[3]For example the following joins of three relations $(A \bowtie B) \bowtie C$ and $A \bowtie (B \bowtie C)$ are represented by only one RelOptInfo

rtekind - distinguishes plain relation, subquery, or function

indexlist - list of structures describing relation's indexes

pages - number of disk pages in relation (zero if not a table)

tuples - number of tuples in relation (not considering restrictions)

subplan - plan for subquery (NULL if it's not a subquery)

We can see, that RelOptInfo contains number of fields with information about the size of the relation, tuples and attributes. This information is intended primarily for use in cost calculation routines so that system catalog access is minimized.

There are several other fields, apart from the fields for holding size of the relation, which deserve discussion. An important part of the RelOptInfo is the list of Path structures. *Path* (also called *access path*) is a low level representation of a query execution plan. It specifies not only the order in which relations should be joined but also which physical operators should be used for each operation in the query. PostgreSQL holds several paths for each RelOptInfo stored in cheapest_startup_path, cheapest_total_path and cheapest_unique_path! as described above.

RestrictInfo structure

RestrictInfo structure represents a restriction clause. PostgreSQL creates a RestrictInfo structure for each AND sub-clause of a query restriction condition in WHERE or JOIN ... ON clause. RestrictInfo structures can be used to filter tuples based on the condition they represent. Another area where RestrictInfo is important is when restriction conditions are moved in a query tree in order to minimize intermediate relation size. This operation has to be used carefully, because operations like outer joins can prevent moving conditions in the query tree. When building join relations, RestrictInfo structures are placed in joininfo lists of the RelOptInfo. Joininfo lists are then used to ensure that generated join orderings are valid.

4.2.2 PostgreSQL routines used by JOS

JOS module uses several routines from the PostgreSQL optimizer. The most important are:

make_join_rel() [4] This function is used to find or create RelOptInfo structure that represents join of two relations supplied as arguments. If the

[4] in src/backend/optimizer/path/joinrels.c

RelOptInfo for the given relations does not exist, new structure is created. This function also adds paths which represent the join of the two relations to the path list in the RelOptInfo. In case the two relations cannot be joined due to join ordering restrictions, the function returns null value.

Memory handling [5] PostgreSQL uses memory manager to allocate and deallocate dynamic memory. Because of this, custom versions of functions to allocate and deallocate memory are used. Allocation is done using palloc() and deallocation using pfree() backend macros. Memory is also divided into contexts and memory is allocated in a given context[6]. This makes it easy to deallocate memory since the whole context can be freed without deallocating all chunks of memory allocated using palloc(). In some cases, JOS requires that memory is allocated in different context and uses methods to switch to a different context. The reason for allocating memory in a different context is, that optimizer routines, such as make_join_rel() modify global planner state stored in PlannerInfo structure. This makes it impossible to execute join ordering algorithms repeatedly, because PlannerInfo contains data from a previous iteration. To overcome this, new memory context is created and modifications to the PlannerInfo are done in this context. Once the operation is finished, contexts are switched back and global data structures are not modified.

4.3 Implementation of the Common module

Many of the implemented algorithms share number of similar operations which were all put in one place. In the JOS module sources the common routines can be found in the `common` directory. The most important of these routines are the ones that generate random solutions and implement the move from one solution to another. They are used in all algorithms with the exception of greedy algorithm and have significant influence on the results.

Solutions generated by the algorithms can be represented as an ordering of relations which can easily be implemented using an array of relation identiers. When the solutions are represented this way, no information about the join tree structure is available, therefore we limit the solution space to left-deep trees. In order to explore also space with bushy-trees, we have decided to split the solution representation into two parts. The first part is ordering of relations just as mentioned before. The second part of the solution is the

[5] src/backend/utils/mmgr/mcxt.c

[6] The following routines for handling memory contexts are used: AllocSetContextCreate(), MemoryContextSwitchTo() and MemoryContextDelete()

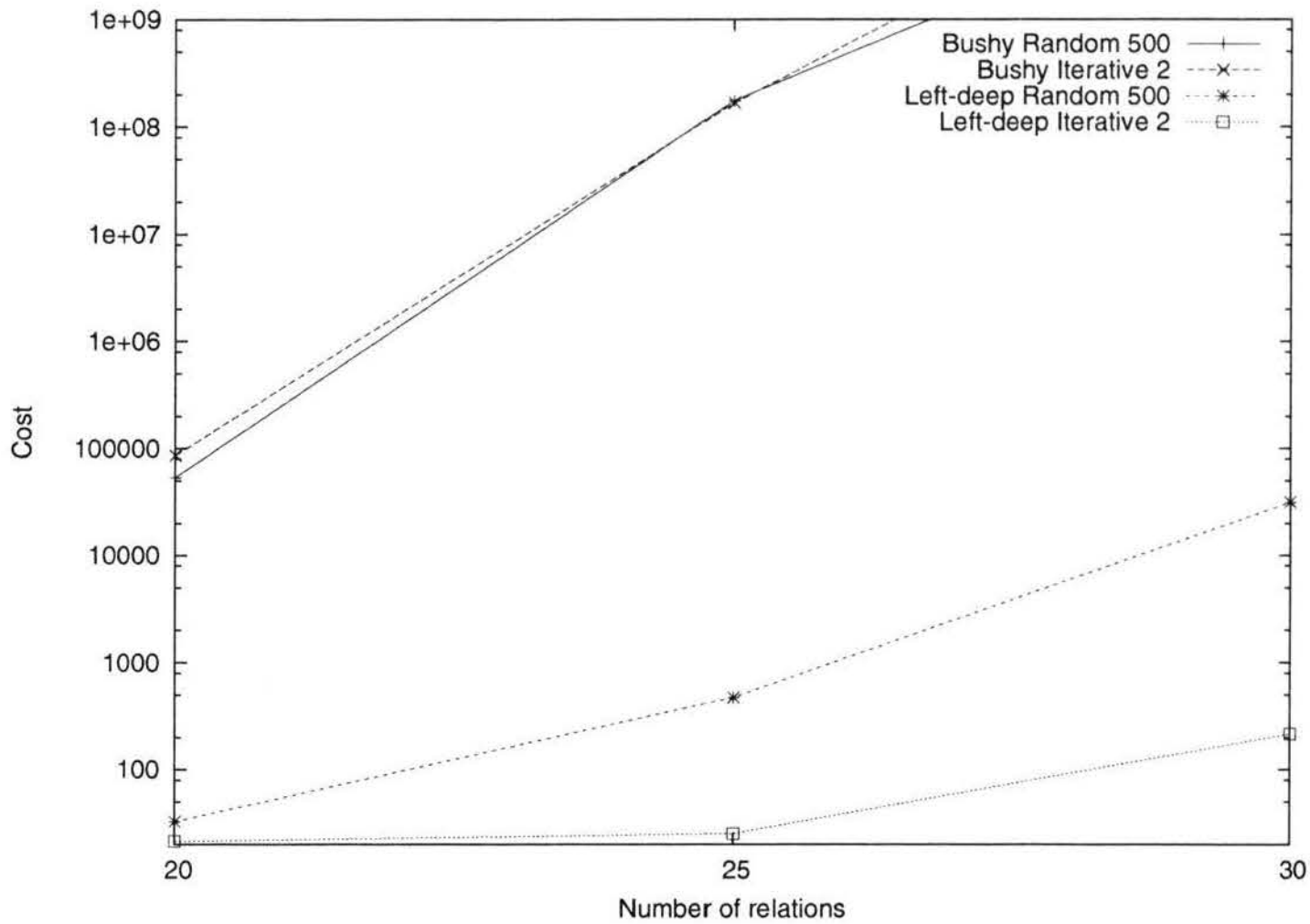


Figure 4.1: Comparison of left-deep and bushy search spaces

structure of the join tree represented using the Prufer code. When generating a solution, we create random tree structure and assign relations to the leaves of the tree in random order. Given an existing solution, neighbour is generated by modifying the ordering of relations assigned to the leaves of the tree.

Despite our efforts, the testing showed that when the algorithms search the space of bushy trees, the cost of the resulting plans is a lot higher than for space with only left-deep trees. Figure 4.1 shows the results of random sampling and iterative improvement algorithms.

Chapter 5

Benchmarks of implemented algorithms

In order to test the implementation of algorithms for finding optimal join ordering we have created a framework for benchmarking which can be used to compare the results of different algorithms. In the first part we will describe the benchmarking framework and second part will give experimental results of implemented algorithms.

5.1 Benchmarking framework

The benchmarking framework is used to perform three operations: generate testing database, generate queries that can be run on this database and automatically run these queries using multiple algorithms.

5.1.1 Generating database and queries for testing

To be able to test queries with large number of joins, we have designed a database with 52 different tables divided into several groups. Due to the fact that data and queries for this database are generated randomly, there are several restrictions that had to be applied. Although the database does not contain real data, it provides sufficient complexity and size for testing optimization algorithms.

The database contains 5 basic types of tables. These types differ in size of the tables as well as in their structure.

Table type	Number of tuples	Number of tables
Tiny	10 - 100	7
Small	100 - 1000	15
Medium	1000 - 10000	15
Large	10000 - 100000	10
Huge	100000 -1000000	5

In each type there are tables with different structure. Data types of the columns are restricted to integer, numeric, varchar and date. Each table has primary key and automatically generated index on the key attribute. Approximately 25 per cent of the columns can contain NULL values. In case that a column can be NULL, approximately 20 per cent of the values are actually NULL. Apart from implicit indexes on primary key columns, there are explicit indexes on 20 per cent of other columns.

Utility to generate the SQL commands for creating database structure is available at the accompanying CD. This utility can also generate data for the database tables in the form of INSERT statements.

Given a database structure, SQL queries can be generated by ainother utility, which is also available on the CD. Currently, two types of queries can be generated:

Star queries - structure of a query graph looks like a star. These queries have one or more central tables, also called fact tables and a large number of outer tables called dimension tables, which are joined to the fact table. Due to the fact that queries with large number of tables need to be generated, there are usually several fact tables (stars) joined together in one query. Number of fact tables can be configured during query generation.

Chain queries - this type of queries consists of tables joined together in a chain, where each of the tables is joined to two others, except for the first and the last one. Large chain queries can consist of several of these chains and the number of chains in one query can be configured during query generation.

Apart from the structure of the query, several other attributes can be specified when queries are generated. These include using left, right or full outer joins, cartesian joins, number of conditions in the where clause or list of columns that should be selected from the result. Documentation of these options and other details can be found in the source code of the query generation utility.

5.2 Experimental results

This section gives an overview of experimental results of all implemented algorithms.

Each algorithm was tested with different parameters and the results are shown in the attached graphs. The goal of testing different versions of the algorithms was to find the best configuration of each algorithm. The testing was performed using queries with 20, 25 and 30 relations and there were 20 different queries of each size. Furthermore, all queries were run 5 times and the average cost was used for comparing the algorithms.

The testing was performed using PostgreSQL 8.3.7 running on the Linux operating system on a computer with 2.8 GHz dual core processor and 3 gigabytes of RAM.

Different versions of each algorithm were compared to the genetic algorithm built into the PostgreSQL. Once we were able to select the best configuration of implemented algorithms, we could compare them to each other. The following sections first compare different versions of each algorithm with the genetic algorithm and then present the comparison of all implemented algorithms.

5.2.1 Greedy algorithm

Implementation of greedy algorithm in JOS follows the general description given in section 2.4.1. The algorithm is initialized by the list of base relations and extends the solution with one relation in each step. The search for the next relation in each step is done by calling functions used in dynamic programming algorithm which are part of the Postgres optimizer. During testing, greedy algorithm had extremely low running time. On the other hand it was not able to find better solutions than the genetic algorithm. The comparison of these two algorithms is shown in Figure 5.1. The greedy algorithm has no configurable parameters and therefore only one version of the algorithm was tested.

5.2.2 Random sampling

Implementation of random sampling algorithm is very straightforward. It generates and evaluates random solutions in a loop and at the end picks the solution with the lowest cost. Several versions of the random sampling algorithm were tested, each with different number of random solutions visited. The comparison for 100, 200, 300, 400 and 500 random solutions is shown in Figure 5.2. No variant of random sampling was able to achieve better results than the genetic optimizer. The running time of random sampling

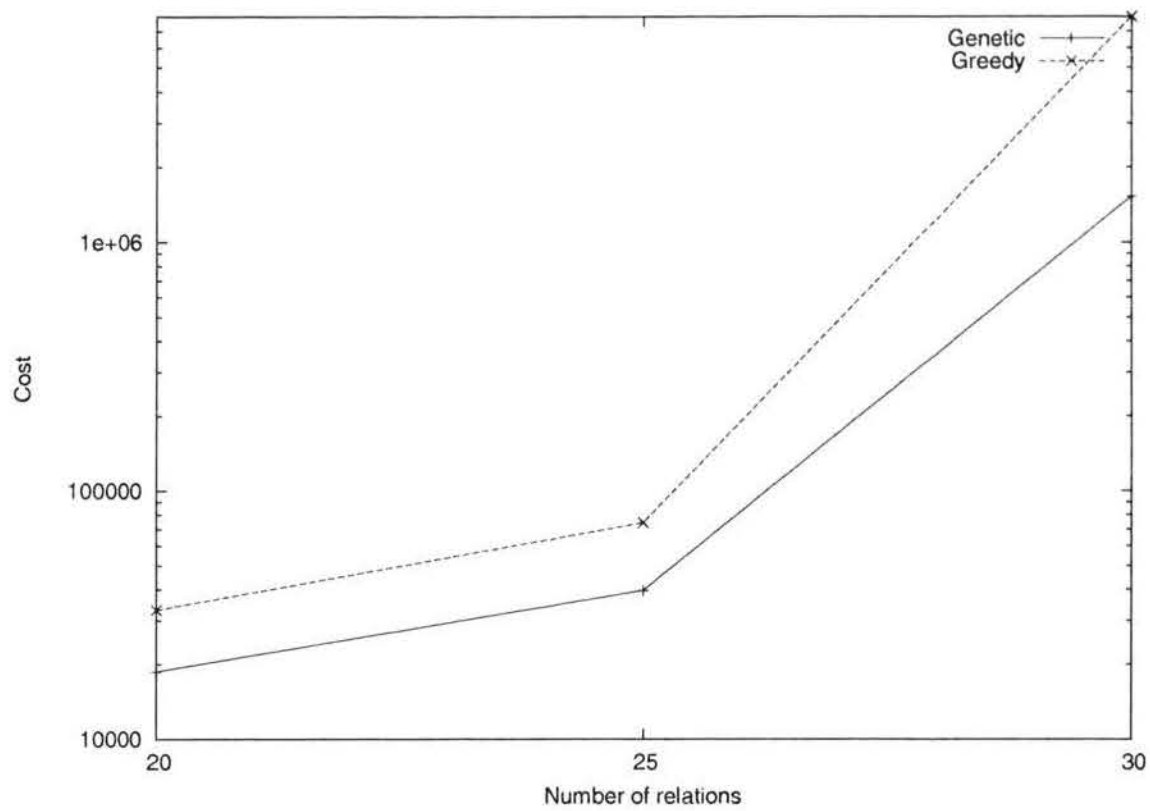


Figure 5.1: Comparison of cost of greedy and genetic algorithm

algorithm was quite small and only the version exploring 500 solutions was slower than genetic optimizer. Detailed information about algorithm running times is in Figure 5.3

5.2.3 Hill-climbing

Hill-climbing algorithm picks a random starting point and searches for a neighbour with lower cost until such a neighbour is found or predefined number of neighbours is visited. The maximum number of visited neighbours can be configured and the algorithm was tested with 50, 75 and 100 neighbours. The results, shown in Figure 5.4, are promising. The hill-climbing algorithm was able to find better solutions than genetic algorithm, even when visiting only 50 neighbours. The running time for this version of algorithm was comparable with the genetic algorithm. Versions of the algorithm visiting more neighbours were able to find even better solutions, although with higher running time. The time spent on optimization of the queries by hill-climbing and genetic algorithm is shown in Figure 5.5.

5.2.4 Iterative improvement

Iterative improvement combines the random sampling and hill-climbing algorithms with one difference. It doesn't move to the first neighbour with the

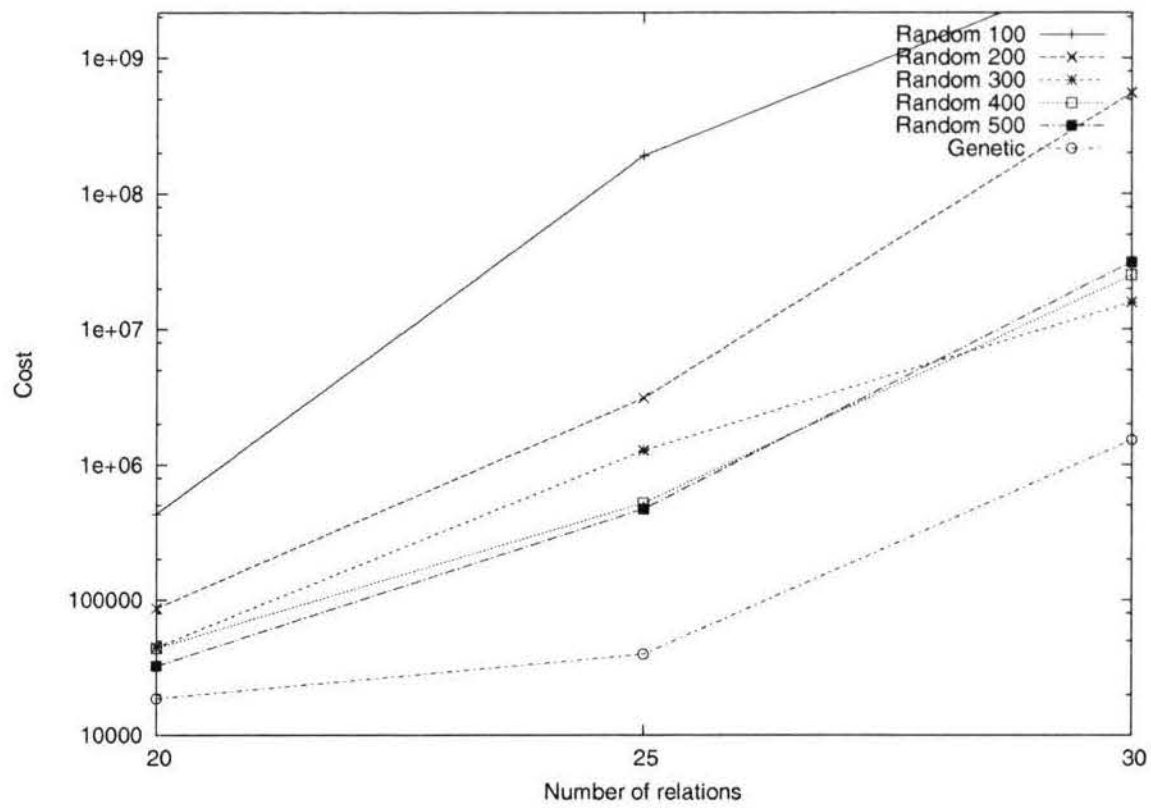


Figure 5.2: Comparison of cost of random sampling and genetic algorithm

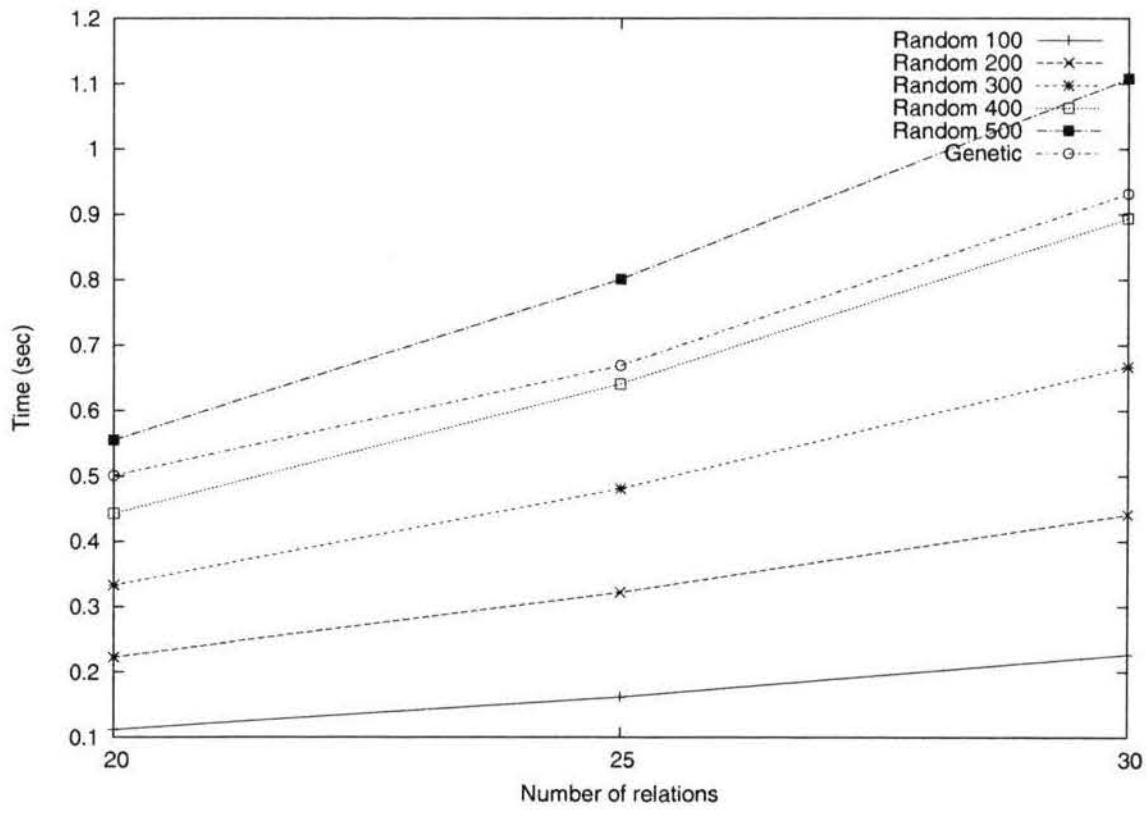


Figure 5.3: Comparison of running time of random sampling and genetic algorithm

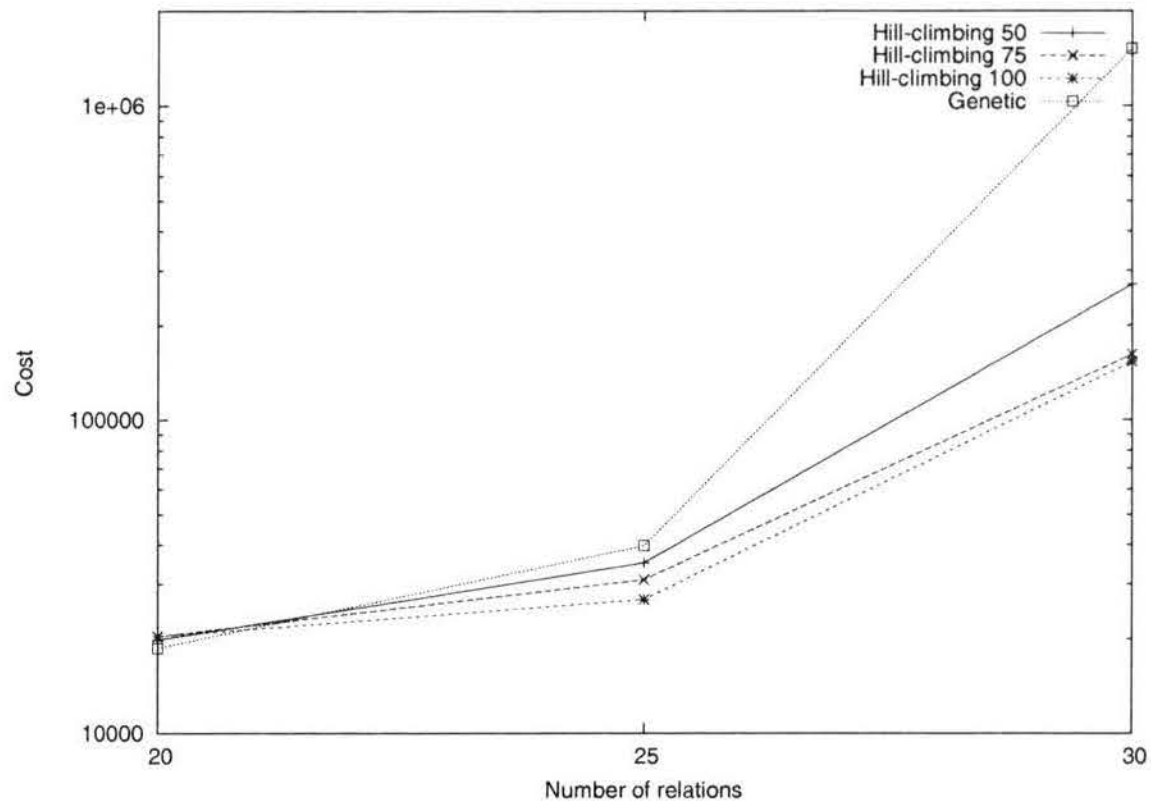


Figure 5.4: Comparison of cost of hill-climbing and genetic algorithm

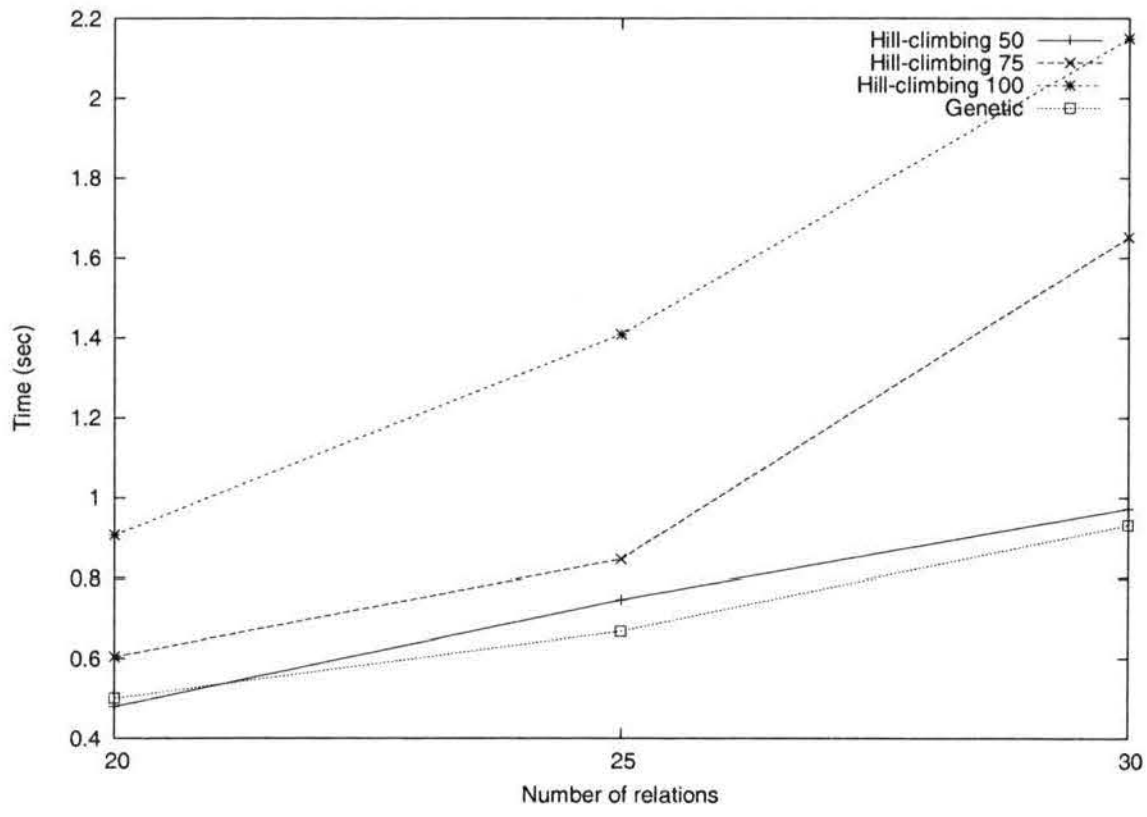


Figure 5.5: Comparison of running time of hill-climbing and genetic algorithm

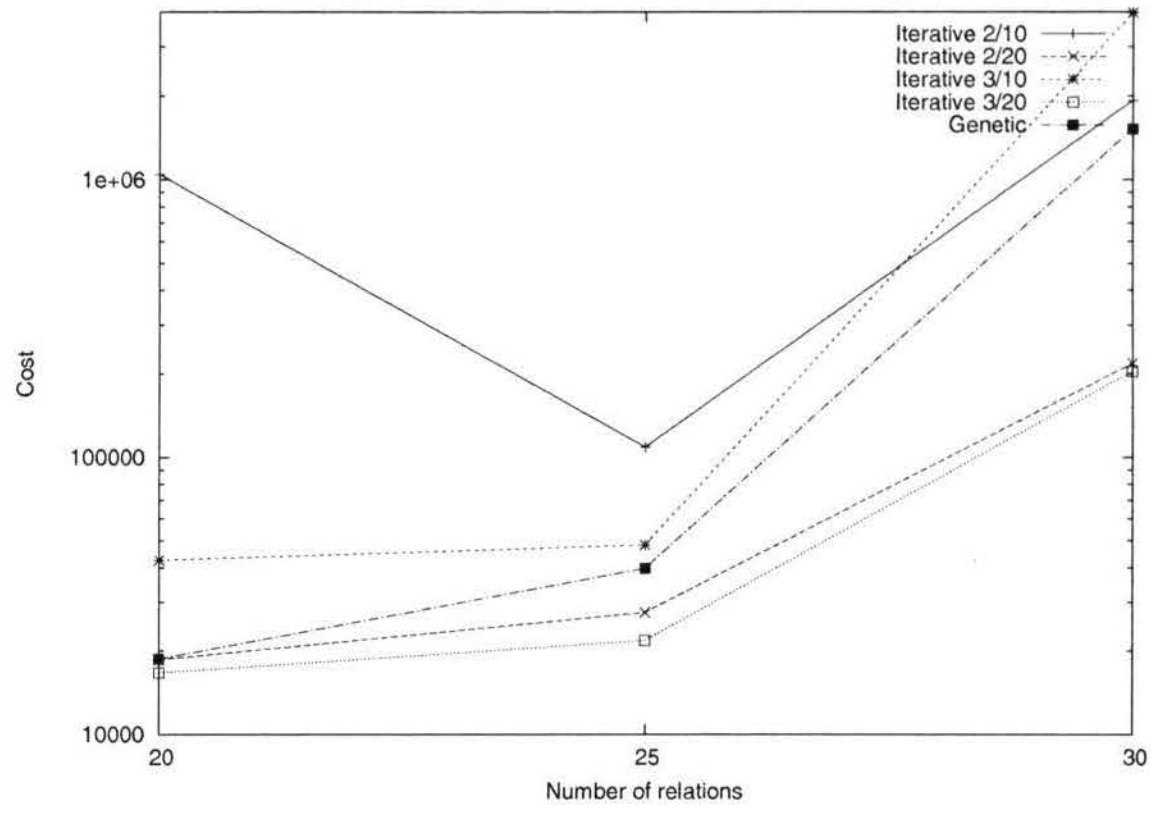


Figure 5.6: Comparison of cost of iterative improvement and genetic algorithm

lower cost but it searches predefined number of neighbours and picks the best one. There are two configurable parameters in iterative improvement algorithm. First is the number of iterations and second is the the maximum number of neighbours visited when searching for a better solution. During testing, iterative improvement was run with 2 and 3 iterations and the maximum number of neighbours was set to 10 and 20. Configurations with higher number of iterations or more neighbours were not considered due to their high running time. The results of different versions of iterative improvement are shown in Figures 5.6 and 5.7 The versions of the algorithm searching 20 neighbours outperformed the genetic algorithm, however they had significantly higher running time. The best results were achieved when the algorithm executed 3 iterations and searched 20 neighbours.

5.2.5 Simulated annealing

The simulated annealing algorithm has the following parameters:

Equilibrium - the value of this parameter was based on the number of relations in the query. The algorithm was tested with two different values of equilibrium: exactly the number of relations in the query and twice the number of relations.

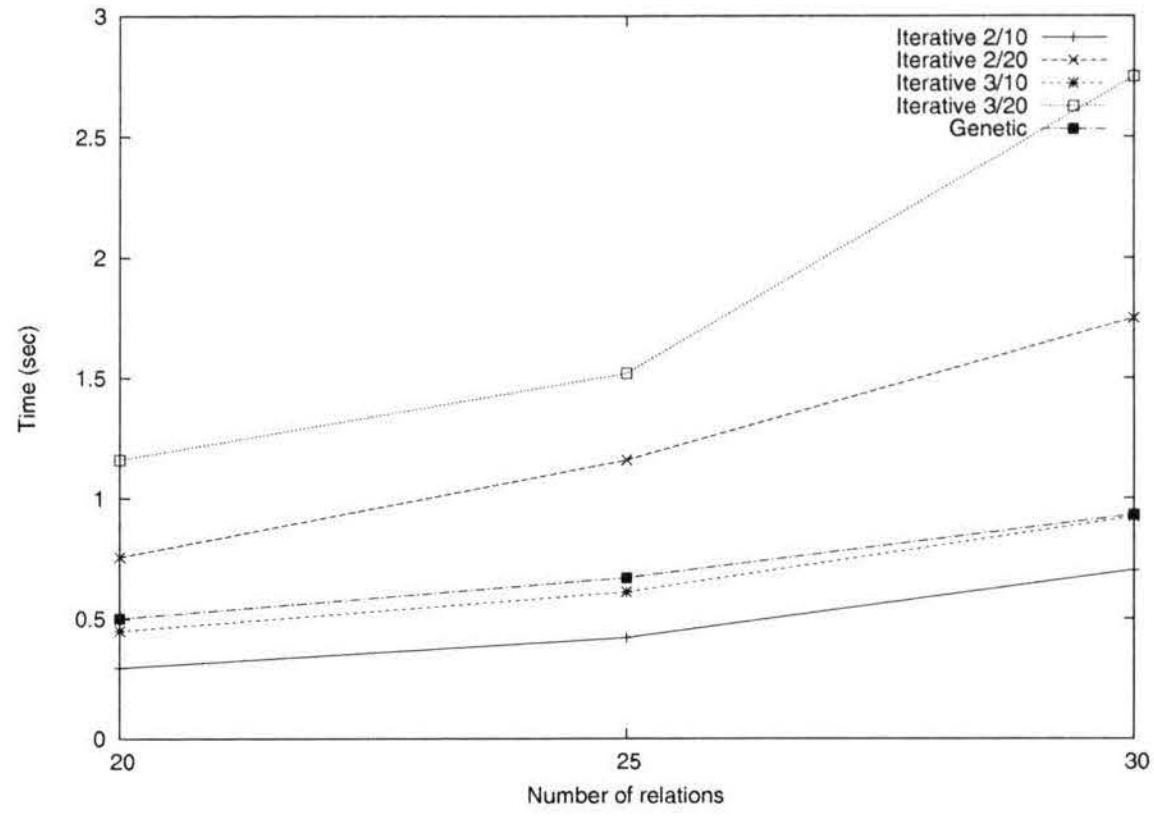


Figure 5.7: Comparison of running time of iterative improvement and genetic algorithm

Initial temperature - initial temeperature was derived from the cost of the initial random solution. With only one initial solution, the performance of the algorithm was poor, regardless of the settings of other parameters. The initial random solution had typically very high cost and therefore the initial temperature was set too high. To overcome this problem, not one but several random solutions were tried and the best one was chosen as the initial solution. Although this increased the running time of the algorithm, the overall performance has improved. The initial temperature was further decreased by taking only one hundredth of the cost of initial solution.

Temperature reduction - the temperature reduction had significant impact on the quality of the results and the running time of the algorithm. Four versions of the algorithm with different temeperature reduction were tested. The values of the parameter were: 0.5, 0.6, 0.7 and 0.8.

Freezing point - The freezing point was not configurable in the current implementation. The algorithm was considered frozen when there were 3 temperature reductions without any improvement of the best solution and the temperature was lower than 1.

Using these parameters and their values, four different version of the algorithm were tested:

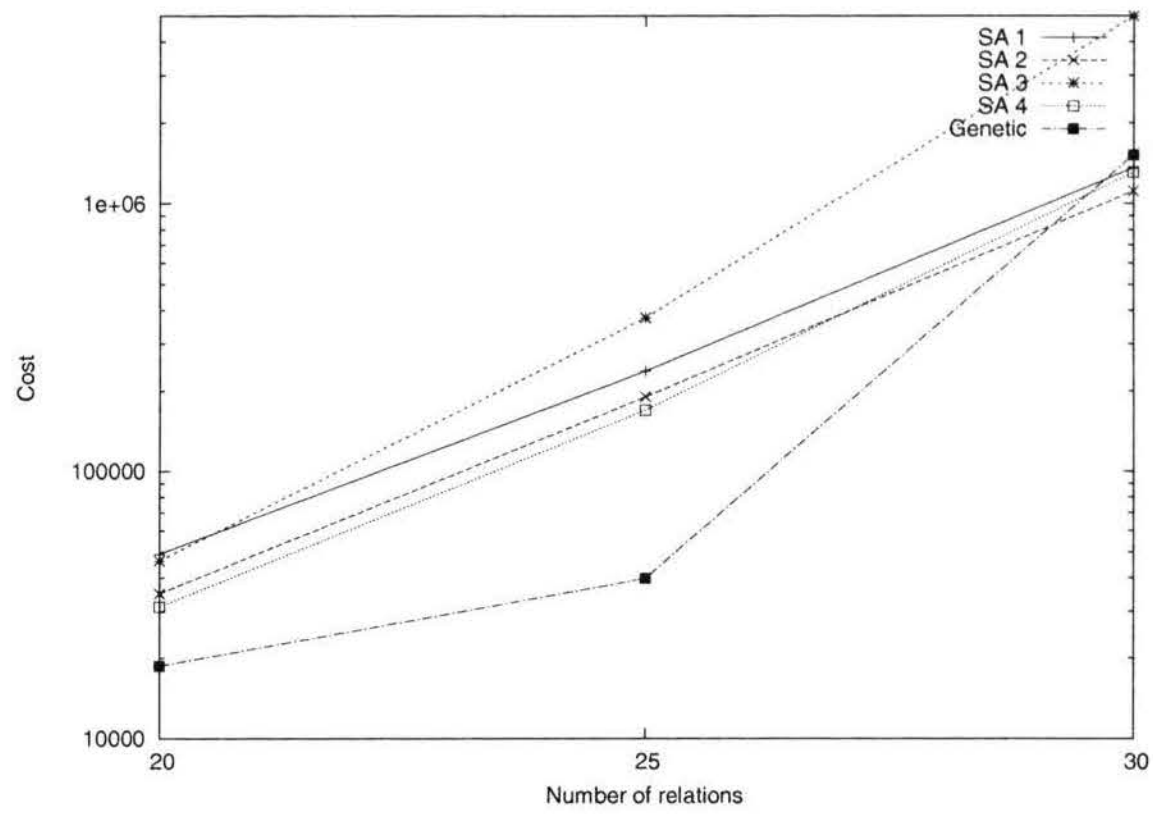


Figure 5.8: Comparison of cost of simulated annealing and genetic algorithm

Version	Equilibrium	Initial temp.	Temp. reduction
1	2 × the number of relations	0.01 × initial cost	0.5
2	2 × the number of relations	0.01 × initial cost	0.6
3	number of relations	0.01 × initial cost	0.7
4	number of relations	0.01 × initial cost	0.8

The performance of the algorithm is shown in Figure 5.8. All version of the algorithm were outperfomed by the genetic algorithm and their running time, shown in Figure 5.9 was also worse.

5.2.6 Two phase optimization

The two phase optimization algorithm combines the iterative improvement and simulated annealing and shares their configuration parameters. In two phase optimization, different values of these parameters were used than for standalone iterative improvement and simulated annealing algorithms to limit the running time. Five different versions of the two phase optimization were considered using the following parameter values:

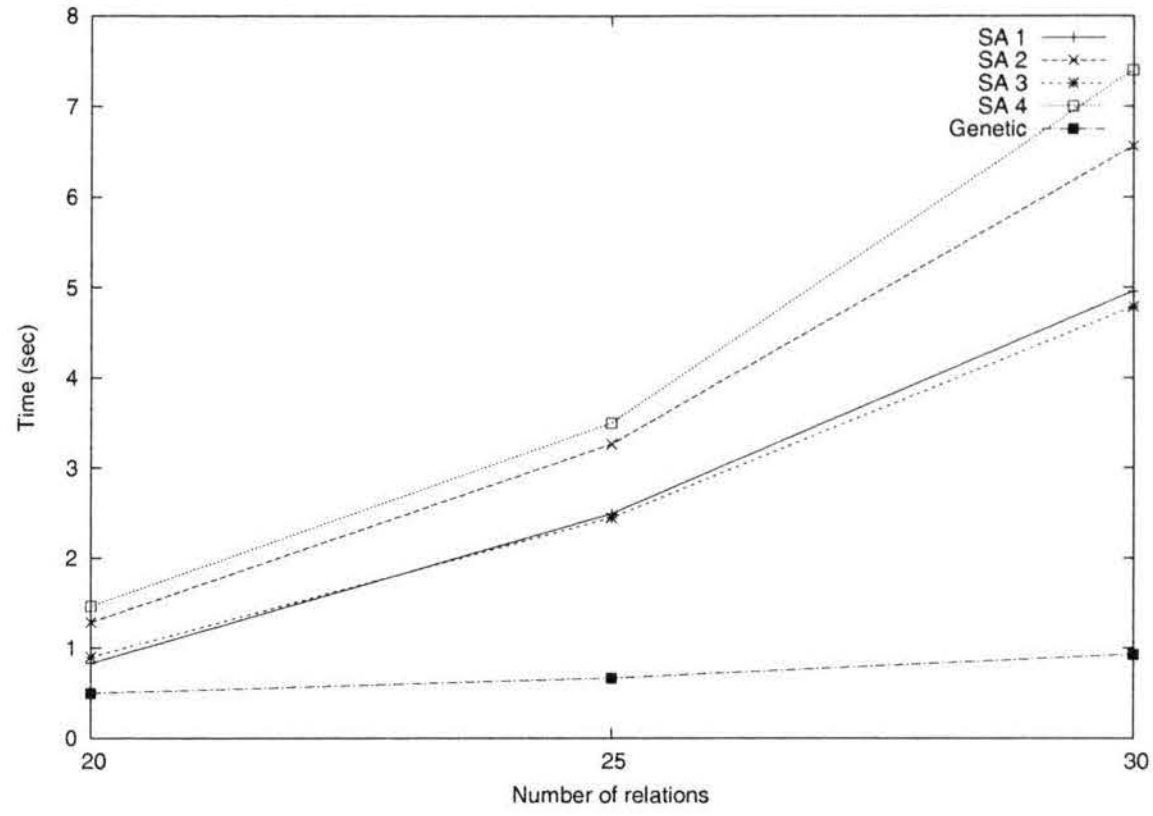


Figure 5.9: Comparison of running time of simulated annealing and genetic algorithm

Version	Iterations (II)	Neighbours (II)	Temp. reduction (SA)
1	2	10	0.3
2	2	10	0.4
3	2	20	0.3
4	2	20	0.4
5	2	10	0.5

Two parameters of the algorithm were the same in all versions:

- Equilibrium of the simulated annealing algorithm was set to the number of relations in the query

- Initial temperature of the simulated annealing algorithm was set to $0.01 \times$ cost of initial solution

The comparison of the cost of different versions of the algorithm is shown in Figure 5.10. Optimization time for all versions of the algorithm is shown in figure 5.11. In terms of the cost of the solution, the second and the fourth version were the best, however the fourth version had the highest running time.

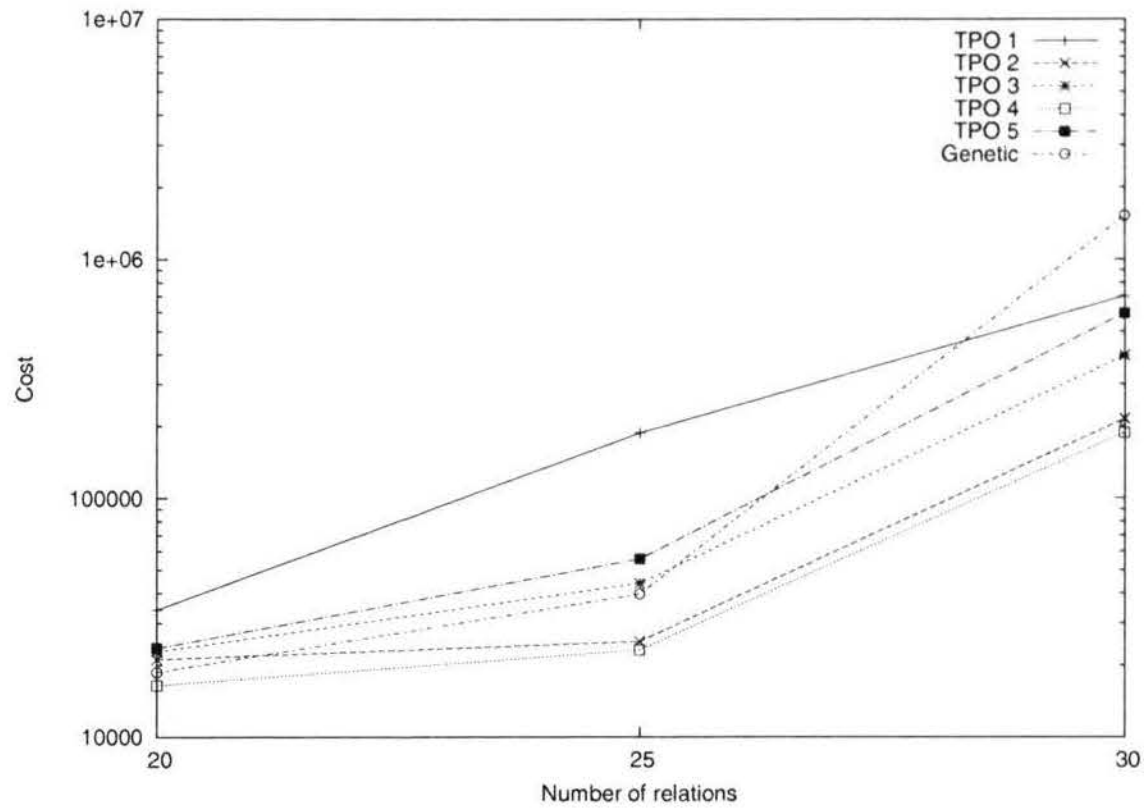


Figure 5.10: Comparison of cost of two phase optimization and genetic algorithm

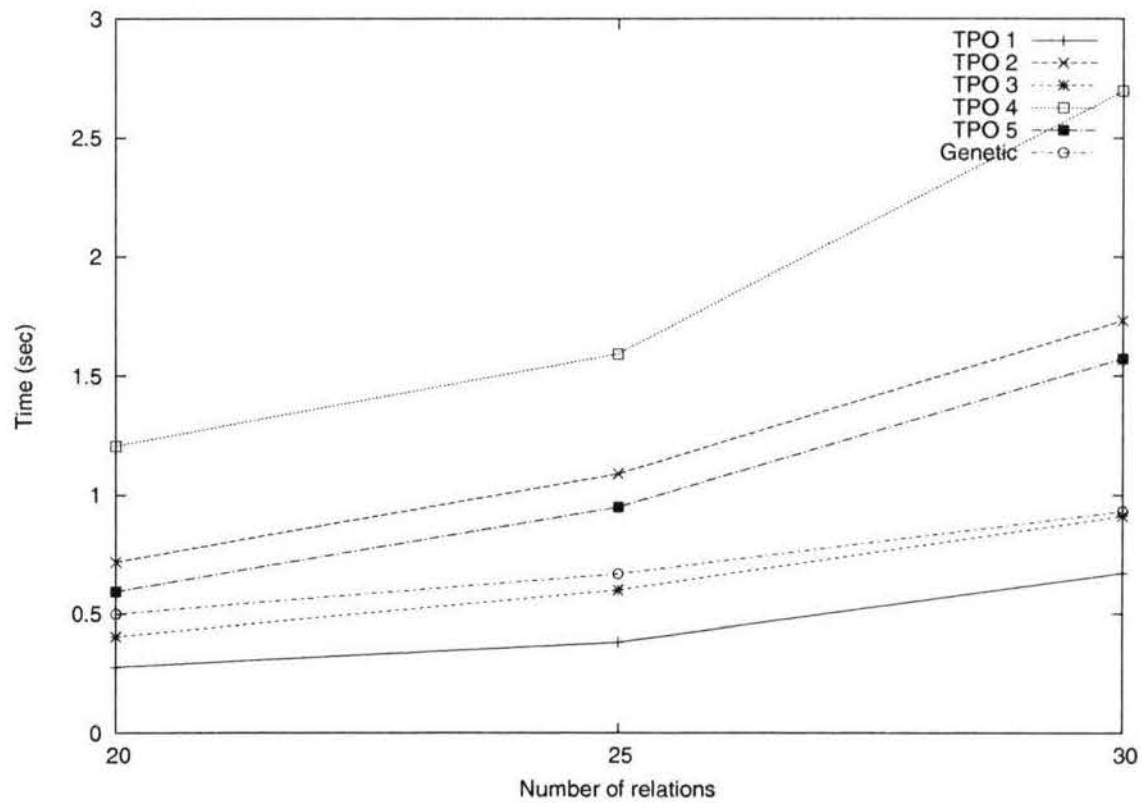


Figure 5.11: Comparison of running time of two phase optimization and genetic algorithm

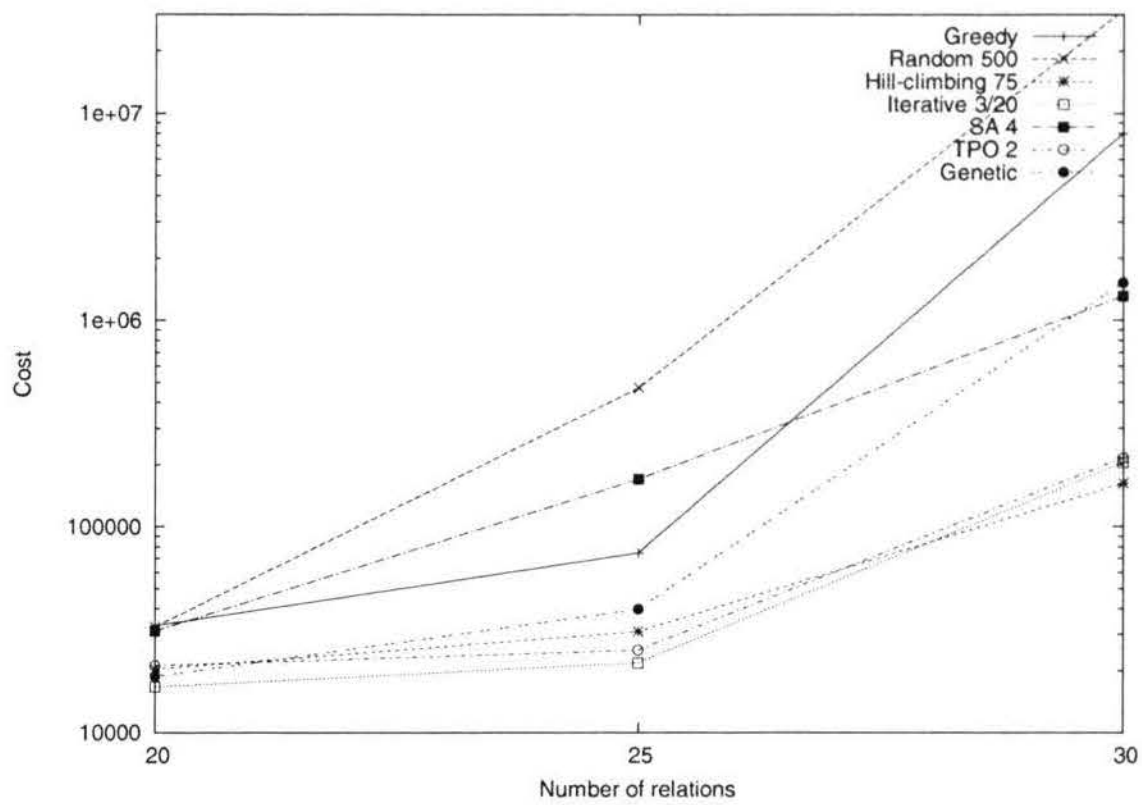


Figure 5.12: Comparison of cost of selected versions of the algorithms

5.2.7 Comparison of selected versions of different algorithms

Based on the results of the algorithms in the previous paragraphs, the following versions of the algorithms were compared to each other:

Greedy algorithm

Random sampling - version examining 500 random solutions

Hill-climbing - version searching up to 75 random neighbours

Iterative improvement - version performing 3 iterations and searching 20 neighbours

Simulated annealing - version number 4

Two phase optimization - version number 2

The comparison of the execution cost of the algorithms and their running time is shown in Figures 5.12 and 5.13 We can clearly see that random sampling algorithm produced the worse execution plans and that more sophisticated algorithms have significantly better performance. The same observation applies for the greedy algorithm. The best execution plans were generated by the iterative improvement (for queries with 20 and 25 tables). The two phase optimization and hill-climbing algorithms found execution

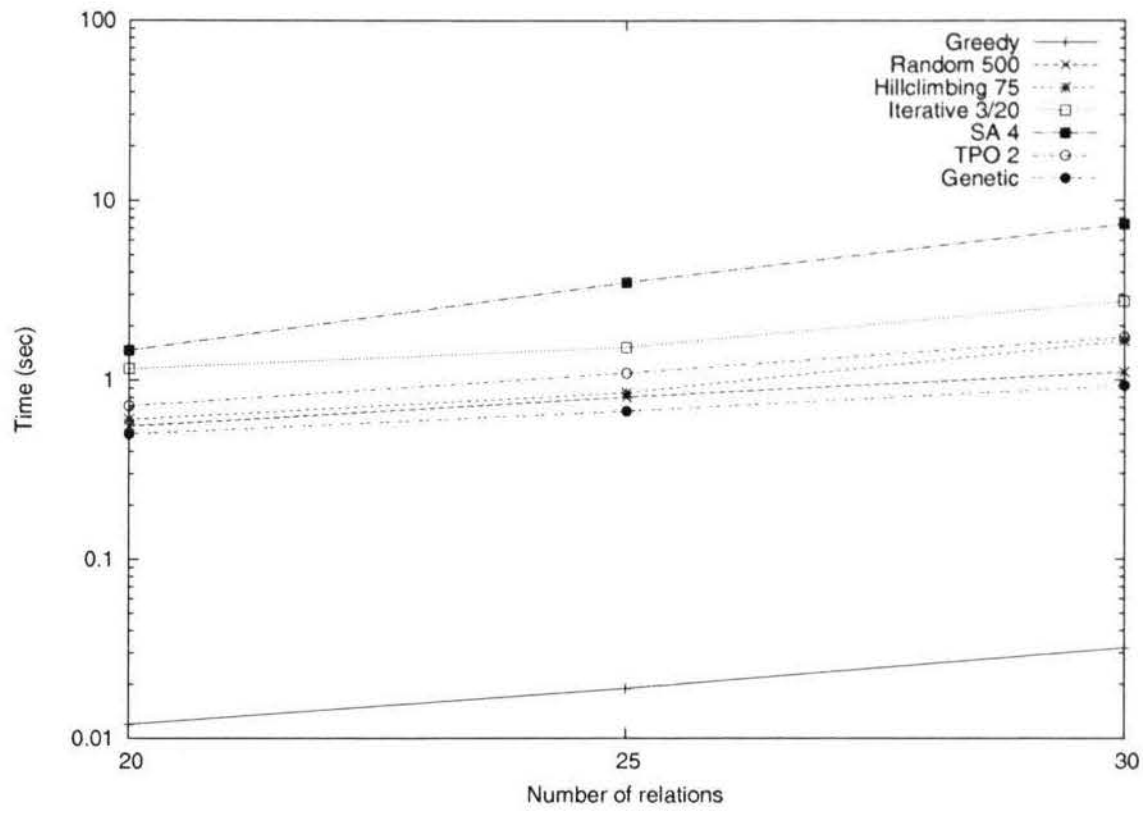


Figure 5.13: Comparison of running time of selected versions of the algorithms

plans with similar cost and for queries with 20 and 25 tables were only a little worse than the iterative improvement algorithm. Genetic algorithm also achieved results close to the iterative improvement, but produced far worse execution plans for queries with 30 relations.

As far as running time of algorithms is concerned, it is clear that the results are influenced by the selection of the versions of algorithms. For most of the algorithms, the running time is similar and there are only small differences. The two outstanding cases are simulated annealing and greedy algorithm. The speed of the greedy algorithm is given by the fact that it doesn't pick random solutions but constructs the solution iteratively and therefore can handle very large queries. The second case is the simulated annealing algorithm, which had significantly higher running time than the other algorithms. The weak overall performance of the simulated annealing might be caused by improper configuration, but there could be other reasons. The two phase optimization algorithm, which uses simulated annealing performed well both in terms of solution cost and running time. Due to the fact, that two phase optimization uses iterative improvement for finding the initial solution, the second phase where simulated annealing algorithm is used has significantly better starting conditions than regular simulated annealing starting from a random solution.

Chapter 6

Conclusion

In this work we gave theoretical overview of the query optimization in relational database systems and examined the implementation of query optimizer of the PostgreSQL in detail. Although database query optimization is studied for a long time, it is still a very active area of research and there is a number of open problems and opportunities for further study.

The implementation part of this work focused on algorithms for finding optimal join ordering in relational database queries. We concentrated on queries involving large number of joins where optimal solution cannot be found in a reasonable time and therefore alternatives to exhaustive search of all possible query execution plans must be considered. Six different algorithms for finding join ordering were implemented as well as framework for integration of these algorithms into the PostgreSQL database system.

The experimental results we obtained show that some of these algorithms perform very well and could be used for improving current query optimizer in PostgreSQL. However, to be widely used much more testing will have to be done using real data and solving real problems.

Appendix A

Instructions for compiling and running JOS module

The accompanying CD contains sources of the JOS module and utilities for generating testing database and queries.

Join Order Search module

Sources of the Join Order Search module are in the `jos` directory. In order to compile the source files, the following programs are necessary:

- compiler of the C language, such as gcc; during development, gcc version 4.3 was used
- PostgreSQL installation including the header files necessary for compiling JOS module. PostgreSQL can be downloaded from `www.postgresql.org`. This page also contains information how to retrieve the sources from the source repository.
- Java Runtime Environment 1.5 or later is required to run utilities for generating database and queries. To compile the source code of these utilities, Java compiler such as javac is necessary.

The JOS module can be compiled and installed using the following commands in the `jos` directory:

```
./configure CPPFLAGS=-I<path_to_postgresql_headers>
make
make install
```

The last step might require that the user is the administrator of the system. PostgreSQL headers can be found in the source distribution of PostgreSQL in the directory `src/include/`. JOS module libraries are typically installed into `/usr/local/lib/postgresql` directory.

In order to test the JOS modules in PostgreSQL the following variables must be set in PostgreSQL configuration file:

```
from_collapse_limit = 100
join_collapse_limit = 100
```

These variables control collapsing of the list of tables in the query to smaller parts, which may be optimized separately. Setting them to a high value will allow for large queries to be processed in one optimizer invocation. PostgreSQL configuration file containing these variables is called `postgresql.conf` and is located in the data directory of PostgreSQL.

JOS module has been developed under the Linux operating system and was not tested on other platforms.

Database and query generation

Utilities for generating the database can be found in the `generatedb` directory. Database and data can be generated using `generatedb.jar` utility. Queries can be generated using `generatequery.jar`. These utilities can be run by:

```
java -jar generatedb.jar
java -jar generatequery.jar
```

Both utilities accept several command line arguments which are displayed by running them without any arguments. There is database definition file in `generatedb` called `dbsetup.txt` that can be used for building the testing database. Directory `generatedb/src` contains sources of the utilities and the whole directory `generatedb` can be opened in Netbeans as a project and built easily using IDE.

List of Figures

Bibliography

[1] Kristin Bennett, Michael C. Ferris, and Yannis E. Ioannidis. A genetic algorithm for database query optimization. In *In Proceedings of the fourth International Conference on Genetic Algorithms*, pages 400–407. Morgan Kaufmann Publishers, 1991.

[2] S. Chaudhuri. An overview of query optimization in relational systems. *Proceedings of the seventeenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, 1998.

[3] EnterpriseDB. Postgresql vs. mysql, a comparison of enterprise suitability. White Paper, May 2008.

[4] César A. Galindo-Legaria, Arjan Pellenkoft, and Martin L. Kersten. Fast, randomized join-order selection - why use transformations? In *VLDB '94: Proceedings of the 20th International Conference on Very Large Data Bases*, pages 85–95, San Francisco, CA, USA, 1994. Morgan Kaufmann Publishers Inc.

[5] J. Widom H. Garcia-Molina, J.D. Ullman. *Database Systems: The Complete Book*. Prentice Hall, Upper Saddle River, New Jersey, 2001.

[6] Wook-Shin Han, Wooseong Kwak, Jinsoo Lee, Guy M. Lohman, and Volker Markl. Parallelizing query optimization. *Proc. VLDB Endow.*, 1(1):188–200, 2008.

[7] Yannis Ioannidis. The history of histograms (abridged). In *Proc. of VLDB Conference*, 2003.

[8] Robert Philip Kooi. *The optimization of queries in relational databases*. PhD thesis, Cleveland, OH, USA, 1980.

[9] A. Kemper M. Steinbrunn, G. Moerkottei. Heuristic and randomized optimization for the join ordering problem. *The VLDB Journal 6, 191-208*, 1997.

[10] Guido Moerkotte and Thomas Neumann. Analysis of two existing and one new dynamic programming algorithm for the generation of optimal bushy join trees without cross products. In *VLDB '06: Proceedings of the 32nd international conference on Very large data bases*, pages 930–941. VLDB Endowment, 2006.

[11] Guido Moerkotte and Thomas Neumann. Dynamic programming strikes back. In *SIGMOD '08: Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 539–552, New York, NY, USA, 2008. ACM.

[12] J. Gehrke R. Ramakrishnan. *Database Management Systems, 2nd edition*. McGraw-Hill, Inc., New York, NY, USA, 1999.

[13] P. Griffiths Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *SIGMOD '79: Proceedings of the 1979 ACM SIGMOD international conference on Management of data*, pages 23–34, New York, NY, USA, 1979. ACM.

[14] R. Rivest T. Cormen, C. Leiserson. *Introduction to algorithms, 2nd edition*. MIT Press, MIT, Cambridge, Massachusetts, USA, 2001.